The New C Standard

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

CHANGES

Commentary

The phrase *at the time of writing* is sometimes used. For this version of the material this time should be taken to mean no later than December 2008.

```
2x Jan 2008  1.1   Integrated in changes made by TC3, required C sentence renumbering.
                   60+ recent references added + associated commentary.
                   A few Usage figures and tables added.
                   Page layout improvements.  Lots of grammar fixes.
 5 Aug 2005  1.0b  Many hyperlinks added.  pdf searching through page 782 speeded up.
                   Various typos fixed (over 70% reported by Tom Plum).
16 Jun 2005  1.0a  Improvements to character set discussion (thanks to Kent Karlsson), margin
                   references, C99 footnote number typos, and various other typos fixed.
30 May 2005  1.0   Initial release.
```

README

-4 This book probably needs one of these.

Commentary

While it was written sequentially, starting at sentence 1 and ending with sentence 2043, readers are unlikely to read it in this way.

At some point you ought to read all of sentence 0 (the introduction).

The conventions used in this book are discussed on the following pages.

There are several ways in which you might approach the material in this book, including the following:

- You have read one or more sentences from the C Standard and want to learn more about them. In this case simply locate the appropriate C sentence in this book, read the associated commentary, and follow any applicable references.
- You want to learn about a particular topic. This pdf is fully searchable. Ok, the search options are not as flexible as those available in a search engine. The plan is to eventually produce separate html versions of each C sentence and its associated commentary. For the time being only the pdf is available.

For anybody planning to print a (double sided) paper copy. Using 80g/m^2 stock produces a stack of paper that is 9.2cm (3.6inches) deep.

Preface

Commentary

This book contains a detailed analysis of the International Standard for the C language,[-3.1] excluding the library from a number of perspectives. The organization of the material is unusual in that it is based on the actual text of the published C Standard. The unit of discussion is the individual sentences from the C Standard (2043 of them).

Readers are assumed to have more than a passing familiarity with C.

C90

My involvement with C started in 1988 with the implementation of a C to Pascal translator (written in Pascal). In 1991 my company was one of the three companies that were joint first, in the world, in having their C compiler formally validated. My involvement with the world of international standards started in 1988 when I represented the UK at a WG14 meeting in Seattle. I continued to head the UK delegation at WG14 meetings for another six years before taking more of a back seat role.

C++

Having never worked on a C++ compiler or spent a significant amount of time studying C++ my view on this language has to be considered as a C only one. While I am a member of the UK C++ panel I rarely attend meetings and have only been to one ISO C++ Standard meeting.

There is a close association between C and C++ and the aim of this subsection is the same as the C90 one: document the differences.

Other Languages

The choice of *other languages* to discuss has been driven by those languages in common use today (e.g., Java), languages whose behavior for particular constructs is very different from C (e.g., Perl or APL), and languages that might be said to have been an early influence on the design of C (mostly BCPL and Algol 68).

The discussion in these subsections is also likely to have been influenced by my own knowledge and biases. Writing a compiler for a language is the only way to get to know it in depth and while I have used many other languages I can only claim to have expertise in a few of them. Prior to working with C I had worked on compilers and source code analyzers for Algol 60, Coral 66, Snobol 4, CHILL, and Pascal. All of these languages might be labeled as imperative 3GLs. Since starting work with C the only other languages I have been involved in at the professional compiler writer level are Cobol and SQL.

Common Implementations

The perceived needs of customers drive translator and processor vendors to design and produce products. The two perennial needs of performance and compatibility with existing practice often result in vendors making design choices that significantly affect how developers interact with their products. The common implementation subsections discuss some the important interactions, primarily by looking at existing implementations and at times research projects (although it needs to be remembered that many of research ideas never make it into commercial products).

I have written code generators for Intel 8086, Motorola 68000, Versal (very similar to the Zilog Z80), Concurrent 3200, Sun SPARC, Motorola 88000, and a variety of virtual machines. In their day these processors have been incorporated in minicomputers or desktop machines. The main hole in my cv. is a complete lack of experience in generating code for DSPs and vector processors (i.e., the discussion is based purely on book learning in these cases).

[-3.1] The document analysed is actually WG14/N1256 (available for public download from the WG14 web site www.open-std.org/jtc1/sc22/wg14/). This document consists of the 1999 version of the ISO C Standard with the edits from TC1, TC2 and TC3 applied to it (plus a few typos corrected).

Coding Guidelines

Writing coding guidelines is a very common activity. Whether these guidelines provide any benefit other than satisfying the itch that caused their author to write them is debatable. My own itch scratchings are based on having made a living, since 1991, selling tools that provide information to developers about possible problems in C source code.

The prime motivating factor for these coding guidelines subsections is money (other coding guideline documents often use technical considerations to label particular coding constructs or practices as *good* or *bad*). The specific monetary aspect of software of interest to me is reducing the cost of source code ownership. Given that most of this cost is the salary of the people employed to work on it, the performance characteristics of human information processing is the prime consideration.

Software developer interaction with source code occurs over a variety of timescales. My own interests and professional experience primarily deals with interactions whose timescale are measured in seconds. For this reason these coding guidelines discuss issues that are of importance over this timescale. While interactions that occur over longer timescales (e.g., interpersonal interaction) are important, they are not the primary focus of these coding guideline subsections. The study of human information processing, within the timescale of interest, largely falls within the field of cognitive psychology and an attempt has been made to underpin the discussion with the results of studies performed by researchers in this field.

The study of software engineering has yet to outgrow the mathematical roots from which it originated. Belief in the mathematical approach has resulted in a research culture where performing experiments is considered to be unimportant and every attempt is made to remove human characteristics from consideration. Industry's insatiable demand for software developers has helped maintain the academic status quo by attracting talented individuals with the appropriate skills away from academia. The end result is that most of the existing academic software engineering research is of low quality and suffers from the problem of being carried out by people who don't have the ability to be mathematicians or the common sense to be practicing software engineers. For this reason the results of this research have generally been ignored.

Existing models of human cognitive processes provide a general framework against which ideas about the mental processes involved in source code comprehension can be tested. However, these cognitive models are not yet sophisticated enough (and the necessary empirical software engineering data is not available) to enable optimal software strategies to be calculated. The general principles driving the discussion that occurs in these coding guidelines subsections include:

1. the more practice people have performing some activity the better they become at performing it.

> *Our attitude towards what we listen to is determined by our habits. We expect things to be said in the ways in which we are accustomed to talk ourselves: things that are said some other way do not seem the same to all but seem rather incomprehensible. . . . Thus, one needs already to have been educated in the way to approach each subject.*
>
> Aristotle Metaphysics book II

   Many of the activities performed during source code comprehension (e.g., reasoning about sequences of events and reading) not only occur in the everyday life of software developers but are likely to have been performed significantly more often in an everyday context. Using existing practice provides a benefit purely because it is existing practice. For a change to existing practice to be worthwhile the total benefit has to be greater than the total cost (which needs to include relearning costs),

2. when performing a task people make implicitly cost/benefit trade-offs. One reason people make mistakes is because they are not willing to pay a cost to obtain more accurate information than they already have (e.g., relying on information available in their head rather expending effort searching for it in the real world). While it might be possible to motivate people to make them more willing pay a greater cost for less benefit the underlying trade-off behavior remains the same,

3. people's information processing abilities are relatively limited and cannot physically be increased (this is not to say that the cognitive strategies used cannot be improved to make the most efficient use of

these resources). In many ways the economics of software development is the economics of human attention.

Usage

Software engineering is an experimental, not a theoretical discipline, and an attempt has been made to base the analysis of C on what software developers and language translators do in practice.

The source code for many of the tools used to extract the information needed to create these figures and tables is available for download from the book's web site.

Measuring the characteristics of software that change over many releases (software evolution) is a relatively new research topic. Software evolution is discussed in a few sentences and any future major revision ought to cover this important topic in substantially more detail.

Table -3.1: Occurrences of various constructs in this book.

Quantity	Kind of information
2,022	C language sentences
1,600	C library paragraphs
1,450	Citations to published books and papers
228	Tables
208	Figures
1,721	Unique cross-reference entries

Acknowledgments

-2 The New C Standard: An economic and cultural commentary

Commentary

Thanks to Sean Corfield (corfield.org) and later Gavin Halliday for many interesting discussions on implementing C90. Also thanks to Clive Feather, the UK C panel, the members of WG14, and my consulting customers who were the source of many insights.

Clive Feather reviewed most of the material in this book. Fred Tydeman reviewed the floating-point material in all subsections. Frank Griswold provided a detailed review of over half of the C++ material. Stephen Parker reviewed a very early draft of some of the coding guidelines. Ken Odgers converted the C99 troff to xml.

Most of the work on the scripts and style sheets/macros used for the layout was done by Vic Kirk. Thanks to the authors of TeXlive, grap, pic, graphviz, and a variety of 'nix based tools.

Marilyn Rash (rrocean@shore.net) copyedited 75% of the material.

Thanks to the librarians of Reading, Surrey, and Warwick Universities for providing access to their collections of Journals. Thanks to all those people who made their papers available online (found via Altavista and later Google and Citeseer).

Post version 1.0

Thanks to the following people for reporting problems in previous versions: David Bremner, Giacomo A. Catenazzi, Cliff Click, Martin Elwin, Jeffrey Haemer, Chris Johansen, Kent Karlsson, Philipp Klaus Krause, Chris Lattner, Jonathan Leffler, Riesch Nicolas, Casey Peel, Jesse Perry, Tom Plum, David Poirier, Arvin Schnell, Ralph Siemsen, Pavel Vozenilek, and Gregory Warnes.

Conventions

information defined here

This is a sentence from WG14/N1124, the number on the inside margin (it would be in a bound book) is the sentence number and this wording has been ~~deleted~~added from/to the wording in C99 by the response to a defect report. -1

Commentary

another sentence

This is some insightful commentary on the above sentence. We might also say something relating to this issue in another sentence (see sentence number and reference heading in the outside margin—it would be in a bound book).

Terms and phrases, such as *blah*, visually appear as just demonstrated.

Rationale

This is a quote from the Rationale document produced by the C Committee to put a thoughtful spin on the wording in the standard.

Various fonts and font-styles are used to denote source code examples (e.g., `a+b*c`), keywords (e.g., **`else`**), syntax terminals (e.g., *`integer-constant`*), complete or partial file names (e.g., `.obj`), programs (e.g., `make`), program options (e.g., `-xs1234`), C Standard identifiers (e.g., `wchar_t`), library functions (e.g., `malloc`) and macros (e.g., `offsetof`).

The headers that appear indented to the left, displayed in a bold Roman font, appear in the C Standard between the two C sentences that they appear between in this book.

C90

This section deals with the C90 version of the standard. Specifically, how it differs from the C99 version of the above sentence. These sections only appear if there is a semantic difference (in some cases the words may have changed slightly, leaving the meaning unchanged).

DR #987

This is the text of a DR (defect report) submitted to the ISO C Standard committee.

Response

The committee's response to this DR is that this question is worth repeating at this point in the book.

This is where we point out what the difference, if any (note the change bar), and what the developer might do, if anything, about it.

C++

1.1p1

This is a sentence from the C++ standard specifying behavior that is different from the above C99 sentence. The 1.1p1 in the outside margin is the clause and paragraph number of this quote in the C++ Standard.

This is where we point out what the difference is, and what the developer might do, if anything, about it. You believed the hype that the two languages are compatible? Get real!

Other Languages

Developers are unlikely to spend their entire professional life using a single language. This section sometimes gives a brief comparison between the C way of doing things and other languages.

Comment received during balloting

We vote against the adoption of the proposed new COBOL standard because we have lost some of our source code and don't know whether the requirements in the proposed new standard would invalidate this source.

Common Implementations

Discussion of how implementations handle the above sentence. For instance, only processors with 17 bit integers can implement this requirement fully (note the text in the outside column—flush left or flush right to the edge of the page—providing a heading that can be referenced from elsewhere). gcc has extensions to support 16 bit processors in this area (the text in the outside margin is pushed towards the outside of the page, indicating that this is where a particular issue is discussed; the text appearing in a smaller point size is a reference to material appearing elsewhere {the number is the C sentence number}).

processors 17 bit

translated invalid program

This is a quote from the document referenced in the outside sidebar.

The New C Standard

Coding Guidelines

General musings on how developers use constructs associated with the above sentence. Some of these sections recommend that a particular form of the construct described in the above sentence not be used.

Cg -1.1 Do it this way and save money.

Dev -1.1 A possible deviation from the guideline, for a described special case.

Rev -1.2 Something to look out for during a code review. Perhaps a issue that requires a trade off among different issues, or that cannot be automated.

Example

An example, in source code of the above sentence.

The examples in this book are generally intended to illustrate some corner of the language. As a general rule it is considered good practice for authors to give examples that readers should follow. Unless stated otherwise, the examples in this book always break this rule.

```
1  struct {float mem;} main(void)
2  {
3  int blah;  /* The /* form of commenting describes the C behavior   */
4             // The // form of commenting describes the C++ behavior
5  }
```

Usage

A graph or table giving the number of occurrences (usually based on this book's benchmark programs) of the constructs discussed in the above C sentence.

Table of Contents

Introduction

0 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.26]) is discouraged.

This International Standard is divided into four major subdivisions:

— preliminary elements (clauses 1–4);
— the characteristics of environments that translate and execute C programs (clause 5);
— the language syntax, constraints, and semantics (clause 6);
— the library facilities (clause 7).

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

The language clause (clause 6) is derived from “The C Reference Manual”.

The library clause (clause 7) is based on the 1984 */usr/group Standard*.

Commentary

defect report 0

This book is about the latest version of the C Standard, ISO/IEC 9899:1999 plus TC1, TC2 and TC3 (these contain wording changes derived from WG14's responses to defect reports). It is structured as a detailed, systematic analysis of that entire language standard (clauses 1–6 in detail; clause 7, the library, is only covered briefly). A few higher-level themes run through all this detail, these are elaborated on below. This book is driven by existing developer practices, not ideal developer practices (whatever they might be). How developers use computer languages is not the only important issue; the writing of translators for them and the characteristics of the hosts on which they have to be executed are also a big influence on the language specification.

Every sentence in the C Standard appears in this book (under the section heading C99). Each of these sentences are followed by a Commentary section, and sections dealing with C90, C++, Other Languages, Common Implementations, Coding Guidelines, Example, and Usage as appropriate. A discussion of each of these sections follows.

culture of C 0

Discussions about the C language (indeed all computer languages), by developers, are often strongly influenced by the implementations they happen to use. Other factors include the knowledge, beliefs and biases (commonly known as folklore, or idiom) acquired during whatever formal education or training developers have had and the culture of the group that they current work within. In an attempt to simplify discussions your author has attempted to separate out these various threads.

Your author has found that a common complaint made about his discussion of C is that it centers on what

the standard says, not on how particular groups of developers use the language. No apology is made for this outlook. There can be no widespread discussion about C until all the different groups of developers start using consistent terminology, which might as well be that of the standard. While it is true that your author's involvement in the C Standards' process and association with other like-minded people has resulted in a strong interest in unusual cases that rarely, if ever, occur in practice, he promises to try to limit himself to situations that occur in practice, or at least only use the more obscure cases when they help to illuminate the meaning or intent of the C Standard.

No apologies are given for limiting the discussion of language extensions. If you want to learn the details of specific extensions, read your vendor's manuals.

Always remember the definitive definition is what the words in the C Standard say. In responding to defect reports the C committee have at times used the phrase *the intent of the Committee*. This phrase has been used when the wording in the standard is open to more than one possible interpretation and where committee members can recall discussions (via submitted papers, committee minutes, or committee email) in which the intent was expressed. The Committee has generally resisted suggestions to rewrite existing, unambiguous, wording to reflect intent (when the wording has been found to specify different behavior than originally intended). 0 defect report

As well as creating a standards document the C committee also produced a rationale. This rationale document provides background information on the thinking behind decisions made by the Committee. Rationale

Wording that appears within a sectioned area like this wording is a direct quote from the rationale (the document used was WG14/N937, dated 17 March 2001).

No standard is perfect (even formally defined languages contain mistakes and ambiguities[710]). There is a mechanism for clarifying the wording in ISO standards, defect reports (DRs as they are commonly called). The text of C99 DRs are called out where applicable. 0 defect report

1 Effort invested in producing the C Standard

The ANSI Committee which produced C90, grew from 13 members at the inaugural meeting, in June 1983, to around 200 members just prior to publication of the first Standard. During the early years about 20 people would attend meetings. There was a big increase in numbers once drafts started to be sent out for public review and meeting attendance increased to 50 to 60 people. Meetings occurred four times a year for six years and lasted a week (in the early years meetings did not always last a week). People probably had to put, say, a week's effort into reading papers and preparing their positions before a meeting. So in round numbers let's say:

```
(20 people × 1.3 weeks × 3 meetings × 1 years) +
(20 people × 1.7 weeks × 4 meetings × 2 years) +
(50 people × 2.0 weeks × 4 meetings × 3 years) ⇒ 1,540 person weeks (not quite 30 years)
```

What about the 140 people not included in this calculation— how much time did they invest? If they spent just a week a year keeping up with the major issues, then we have 16 person years of effort. On top of this we have the language users and implementors reviewing drafts that were made available for public review. Not all these sent in comments to the Committee, but it is not hard to imagine at least another 4 person years of effort. This gives the conservative figure of 50 person years of effort to produce C90.

Between the publication of C90 and starting work on the revision of C99, the C committee met twice a year for three days; meeting attendance tended to vary between 10 and 20. There was also a significant rise in the use of email during this period. There tended to be less preparation work that needed to be done before meetings— say 2 person years of effort.

The C99 work was done at the ISO level, with the USA providing most of the active committee membership. The Committee met twice a year for five years. Membership numbers were lower, at about 20 per meeting.

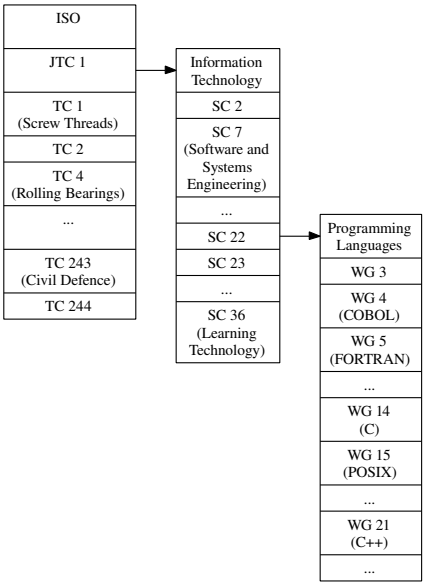


Figure 0.1: The ISO Technical Committee structure— JTC (Joint Technical Committee, with the IEC in this case), TC (Technical Committee), SC (Standards Committee), WG (Working Group).

This gives a figure of 8 person years. During development of C99 there was a significant amount of discussion on the C Standard's email list; just a week per year equates to more than 2 person years (the UK and Japanese national bodies had active working groups, many of whose members did not attend meetings).

Adding these numbers up gives a conservative total of 62 person years of effort that was invested in the C99 document. This calculation does not include the cost of travelling or any support cost (the document duplication bill for one committee mailing was approximately $5,000).

ISO

The C committee structure

The three letters ISO are said to be derived from the Greek *isos*, meaning "the same" (the official English term used is International Organization for Standardization, not a permutation of these words that gives the ordering ISO). Countries pay to be members of ISO (or to be exact, standards organizations in different countries pay). The size of the payment depends on a country's gross domestic product (a measure of economic size) and the number of ISO committees they want to actively participate in. Within each country, standards' bodies (there can be more than one) organize themselves in different ways. In many countries it is possible for their national standards' body(s) to issue a document as a standard in that country. The initial standards work on C was carried out by one such national body — ANSI (American National Standards Institute). The document they published was only a standard in the USA. This document subsequently went
X3J11 0 through the process to become an International Standard. As of January 2003, ISO has 138 national standards bodies as members, a turnover of 150 million Swiss Francs, and has published 13,736 International Standards (by 188 technical committees, 550 subcommittees, and 2,937 working groups)(see Figure 0.1).

The documents published by ISO may be formally labeled as having a particular status. These labels include Standard, Technical Report (Type 1, 2, or 3), and a draft of one of these kinds of documents (there are also various levels of draft). The documents most commonly seen by the public are Standards and Type 2 Technical Reports. A Type 2 Technical Report (usually referred to as simply a TR) is a document that is believed to be worth publishing as an ISO Standard, but the material is not yet sufficiently mature to be published as a standard. It is a kind of standard in waiting.

C90

C90 was the first version of the C Standard, known as ISO/IEC 9899:1990(E) (Ritchie[1169] gives a history of prestandard development). It has now been officially superseded by C99. The C90 sections ask the question: What are the differences, if any, between the C90 Standard and the new C99 Standard?
Text such this occurs (with a bar in the margin) when a change of wording can lead to a developer visible change in behavior of a program.

Possible differences include:

- C90 said X was black, C99 says X is white.
- C99 has relaxed a requirement specified in C90.
- C99 has tightened a requirement specified in C90.
- C99 contains a construct that was not supported in C90.

If a construct is new in C99 this fact is only pointed out in the first sentence of any paragraph discussing it. This section is omitted if the wording is identical (word for word, or there are minor word changes that do not change the semantics) to that given in C99. Sometimes sentences have remained the same but have changed their location in the document. Such changes have not been highlighted.

The first C Standard was created by the US ANSI Committee X3J11 (since renamed as NCITS J11). This document is sometimes called C89 after its year of publication as an ANSI standard (The shell and utilities portion of POSIX[636] specifies a c89 command, even although this standard references the ISO C Standard, not the ANSI one.). The published document was known as ANSI X3.159–1989. X3J11

This ANSI standard document was submitted, in 1990, to ISO for ratification as an International Standard. Some minor editorial changes needed to be made to the document to accommodate ISO rules (a sed script was used to make the changes to the troff sources from which the camera-ready copy of the ANSI and ISO standards was created). For instance, the word Standard was replaced by International Standard and some major section numbers were changed. More significantly, the Rationale ceased to be included as part of the document (and the list of names of the committee members was removed). After publication of this ISO standard in 1990, ANSI went through its procedures for withdrawing their original document and adopting the ISO Standard. Subsequent purchasers of the ANSI standard see, for instance, the words International Standard not just Standard.

2 Updates to C90

Part of the responsibility of an ISO Working Group is to provide answers to queries raised against any published standard they are responsible for. During the early 1990s, the appropriate ISO procedure seemed to be the one dealing with defects, and it was decided to create a Defect Report log (entries are commonly known as *DR*s). These procedures were subsequently updated and defect reports were renamed *interpretation requests* by ISO. The C committee continues to use the term *defect* and DR, as well as the new term *interpretation request*. defect report

Standards Committees try to work toward a publication schedule. As the (self-imposed) deadline for publication of the C Standard grew nearer, several issues remained outstanding. Rather than delay the publication date, it was agreed that these issues should be the subject of an Amendment to the Standard. The purpose of this Amendment was to address issues from Denmark (readable trigraphs), Japan (additional support for wide character handling), and the UK (tightening up the specification of some constructs whose wording was considered to be ambiguous). The title of the Amendment was *C Integrity*.

As work on DRs (this is how they continue to be referenced in the official WG14 log) progressed, it became apparent that the issues raised by the UK, to be handled by the Amendment, were best dealt with via these same procedures. It was agreed that the UK work item would be taken out of the Amendment and converted into a series of DRs. The title of the Amendment remained the same even though the material that promoted the choice of title was no longer included within it.

To provide visibility for those cases in which a question had uncovered problems with wording in the published standard the Committee decided to publish collections of DRs. The ISO document containing such corrections is known as a Technical Corrigendum (*TC*) and two were published for C90. A TC is normative and contains edits to the existing standard's wording only, not the original question or any rationale behind the decision reached. An alternative to a TC is a Record of Response (*RR*), a non-normative document.

Wording from the Amendment, the TCs and decisions on defect reports that had not been formally published were integrated into the body of the C99 document.

A determined group of members of X3J11, the ANSI Committee, felt that C could be made more attractive to numerical programmers. To this end it was agreed that this Committee should work toward producing a technical report dealing with numerical issues.

NCEG

The Numerical C Extensions Group (*NCEG*) was formed on May 10, 1989; its official designation was X3J11.1. The group was disbanded on January 4, 1994. The group produced a number of internal, committee reports, but no officially recognized Technical Reports were produced. Topics covered included: compound literals and designation initializers, extended integers via a header, complex arithmetic, restricted pointers, variable length arrays, data parallel C extensions (a considerable amount of time was spent on discussing the merits of different approaches), and floating-point C extensions. Many of these reports were used as the base documents for constructs introduced into C99.

base document

Support for parallel threads of execution was not addressed by NCEG because there was already an ANSI Committee, X3H5, working toward standardizing a parallelism model and Fortran and C language bindings to it.

C++

Many developers view C++ as a superset of C and expect to be able to migrate C code to C++. While this book does not get involved in discussing the major redesigns that are likely to be needed to make effective use of C++, it does do its best to dispel the myth of C being a subset of C++. There may be a language that is common to both, but these sections tend to concentrate on the issues that need to be considered when translating C source using a C++ translator.

What does the C++ Standard, ISO/IEC 14882:1998(E), have to say about constructs that are in C99?

- *Wording is identical.* Say no more.
- *Wording is similar.* Slight English grammar differences, use of terminology differences and other minor issues. These are sometimes pointed out.
- *Wording is different but has the same meaning.* The sequence of words is too different to claim they are the same. But the meaning appears to be the same. These are not pointed out unless they highlight a C++ view of the world that is different from C.
- *Wording is different and has a different meaning.* Here the C++ wording is quoted, along with a discussion of the differences.
- *No C++ sentence can be associated with a C99 sentence.* This often occurs because of a construct that does not appear in the C++ Standard and this has been pointed out in a previous sentence occurring before this derived sentence.

There is a stylized form used to comment source code associated with C— `/* behavior */`— and C++— `// behavior`.

The precursor to C++ was known as C with Classes. While it was being developed C++ existed in an environment where there was extensive C expertise and C source code. Attempts by Stroustrup to introduce incompatibilities were met by complaints from his users.[1310]

The intertwining of C and C++, in developers mind-sets, in vendors shipping a single translator with a language selection option, and in the coexistence of translation units written in either language making up one program means that it is necessary to describe any differences between the two.

The April 1989 meeting of WG14 was asked two questions by ISO: (1) should the C++ language be standardized, and (2) was WG14 the Committee that should do the work? The decision on (1) was very close, some arguing that C++ had not yet matured sufficiently to warrant being standardized, others arguing that working toward a standard would stabilize the language (constant changes to its specification and implementation were causing headaches for developers using it for mission-critical applications). Having agreed that there should be a C++ Standard WG14 was almost unanimous in stating that they were not the Committee that should create the standard. During April 1991 WG21, the ISO C++ Standard's Committee was formed; they met for the first time two months later.

In places additional background information on C++ is provided. Particularly where different concepts, or terminology, are used to describe what is essentially the same behavior.

In a few places constructs available in C++, but not C, are described. The rationale for this is that a C developer, only having a C++ translator to work with, might accidentally use a C++ construct. Many C++ translators offer a C compatibility mode, which often does little more than switch off support for a few C++ constructs. This description may also provide some background about why things are different in C++.

Everybody has a view point, even the creator of C++, Bjarne Stroustrup. But the final say belongs to the standards' body that oversees the development of language standards, SC22. The following was the initial position.

Resolutions Prepared at the Plenary Meeting of

ISO/IEC JTC 1/SC22

Vienna, Austria

September 23–29, 1991

Resolution AK Differences between C and C++

Notwithstanding that C and C++ are separate languages, ISO/IEC JTC1/SC22 directs WG21 to document differences in accordance with ISO/IEC TR 10176.

Resolution AL WG14 (C) and WG21 (C++) Coordination

While recognizing the need to preserve the respective and different goals of C and C++, ISO/IEC JTC1/SC22 directs WG14 and WG21 to ensure, in current and future development of their respective languages, that differences between C and C++ are kept to the minimum. The word "differences" is taken to refer to strictly conforming programs of C which either are invalid programs in C++ or have different semantics in C++.

This position was updated after work on the first C++ Standard had been completed, but too late to have any major impact on the revision of the C Standard.

Resolutions Prepared at the Eleventh Plenary Meeting of

ISO/IEC JTC 1/SC22

Snekkersten, Denmark

August 24–27, 1998

Resolution 98-6: Relationship Between the Work of WG21 and that of WG14

Recognizing that the user communities of the C and C++ languages are becoming increasingly divergent, ISO/IEC JTC 1/SC22 authorizes WG21 to carry out future revisions of ISO/IEC 14882:1998 (Programming Language C++) without necessarily adopting new C language features contained in the current revision to ISO/IEC 9899:1990 (Programming Language C) or any future revisions thereof.

ISO/IEC JTC 1/SC22 encourages WG14 and WG21 to continue their close cooperation in the future.

Other Languages

Why are other languages discussed in this book? Developers are unlikely to spend their entire working life using a single language (perhaps some Cobol and Fortran programmers may soon achieve this).

C is not the only programming language in the world (although some developers act as if it were). Characteristics of other languages can help sharpen a developer's comprehension of the spirit (design, flavor, world-view) of C. Some of C's constructs could have been selected in several alternative ways, others interrelate to each other.

The functionality available in C can affect the way an algorithm is coded (not forgetting individual personal differences[1117, 1118]). Sections of source may only be written that way because that is how things are done in C; they may be written differently, and have different execution time characteristics,[1119] in other languages. Appreciating the effects of C language features in the source they write can be very difficult for developers to do; rather like a fish trying to understand the difference between water and dry land.

Some constructs are almost universal to all programming languages, others are unique to C (and often C++). Some constructs are common to a particular class of languages— algorithmic, functional, imperative, formal, and so on. The way things are done in C is not always the only way of achieving the same result, or the same algorithmic effect. Sometimes C is unique. Sometimes C is similar to what other languages do. Sometimes there are languages that do things very differently from C, either in implementing the same idea, or in having a different view of the world.

It is not the intent to claim that C or any other language is better or worse because it has a particular design philosophy, or contains a particular construct. Neither is this subsection intended as a survey of what other languages do. No attempt is made to discuss any other language in any way apart from how it is similar or different from C. Other languages are looked at from the C point of view.

Developers moving from C to another language will, for a year or so (or longer depending on the time spent using the new language), tend to use that language in a C-like style (much the same as people learning English tend to initially use the grammar and pronunciations of their native language; something that fluent speakers have no trouble hearing).

Your author's experience with many C developers is that they tend to have a *C is the only language worth knowing attitude*. This section is unlikely to change that view and does not seek to. Some knowledge of how other languages do things never hurt.

There are a few languages that have stood the test of time, Cobol and Fortran for example. While Pascal and Ada may have had a strong influence on the thinking about how to write maintainable, robust code, they have come and gone in a relatively short period of time. At the time of this writing there are six implementations of Ada 95. A 1995 survey[590] of language usage found 49.5 million lines of Ada 83 (C89 32.5 million, other languages 66.18 million) in DoD weapon systems. The lack of interest in the Pascal standard is causing people to ask whether it should be withdrawn as a recognized standard (ISO rules require that a standard be reviewed every five years). The Java language is making inroads into the embedded systems market (the promise of it becoming the lingua franca of the Internet does not seem to have occurred). It is also trendy, which keeps it in the public eye. Lisp continues to have a dedicated user base 40 years after its creation. A paper praising its use, over C, has even been written.[411]

The references for the other languages mentioned in this book are: Ada,[645] Algol 68,[1407] APL,[654] BCPL,[1162] CHILL,[656] Cobol,[634] Fortran,[640] Lisp[650] (Scheme[719]), Modula-2,[646] Pascal,[639] Perl,[1438] PL/1,[633] Snobol 4,[521] and SQL.[641]

References for the implementation of languages that have significant differences from C include APL,[178] functional languages,[1082] and ML.[50]

Common Implementations

3 Introduction

This subsection gives an overview of translator implementation issues. The specific details are discussed in the relevant sentence. The following are the main issues.

- *Translation environment.* This environment is defined very broadly here. It not only includes the language specification (dialects and common extensions), but customer expectations, known translation

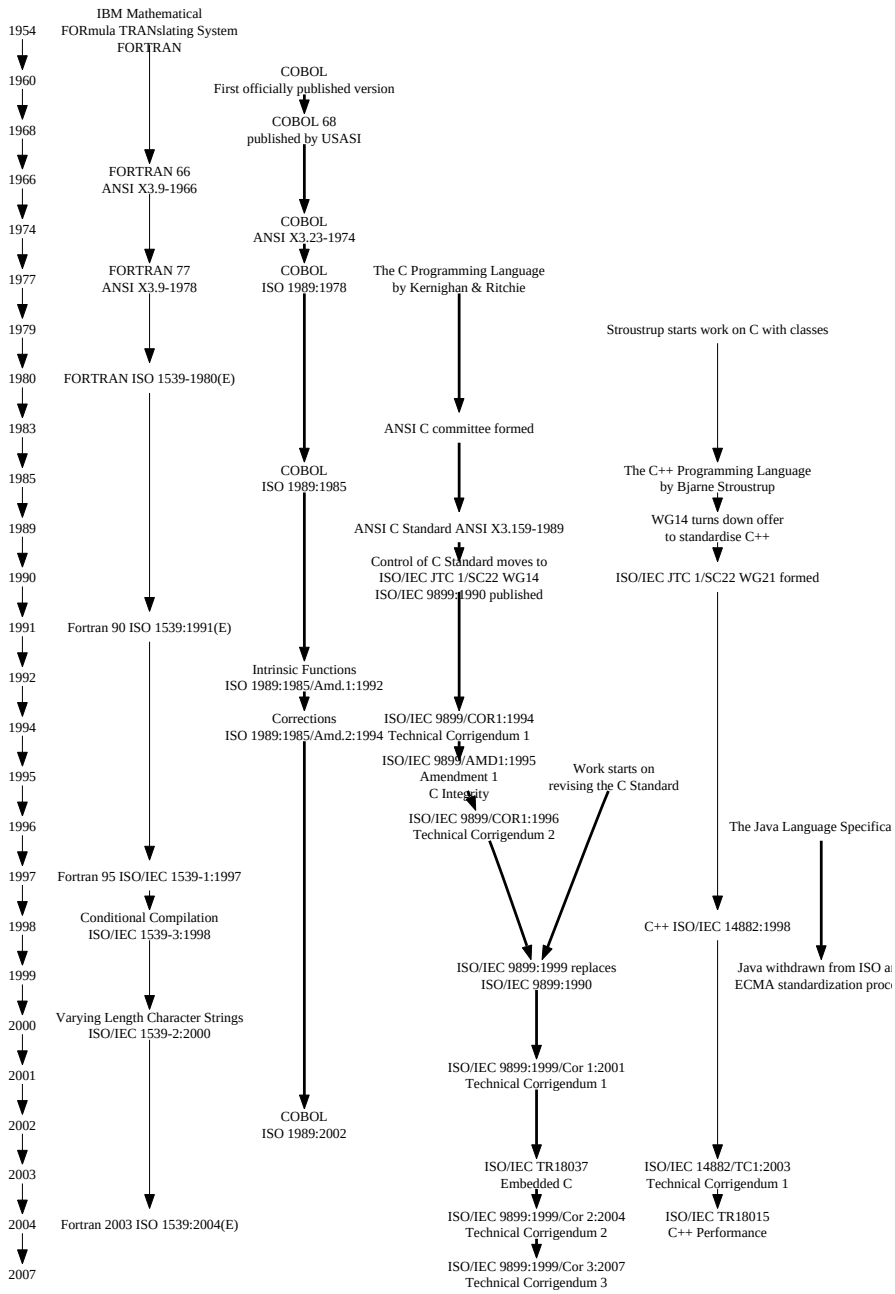


Figure 0.2: Outline history of the C language and a few long-lived languages. (Backus[69] describes the earliest history of Fortran.)

technology and the resources available to develop and maintain translators. Like any other application development project, translators have to be written to a budget and time scale.

- *Execution environment.* This includes the characteristics of the processor that will execute the program image (instruction set, number of registers, memory access characteristics, etc.), and the runtime interface to the host environment (storage allocation, function calling conventions, etc.).
- *Measuring implementations.* Measurements on the internal working of translators is not usually published. However, the execution time characteristics of programs, using particular implementations, is of great interest to developers and extensive measurements are made (many of which have been published).

4 Translation environment

The translation environment is where developers consider their interaction with an implementation to occur. Any requirement that has existed for a long period of time (translators, for a variety of languages, have existed for more than 40 years; C for 25 years) establishes practices for how things should be done, accumulates a set of customer expectations, and offers potential commercial opportunities.

Although the characteristics of the language that need to be translated have not changed significantly, several other important factors have changed. The resources available to a translator have significantly increased and the characteristics of the target processors continue to change. This increase in resources and need to handle new processor characteristics has created an active code optimization research community.

4.1 Developer expectations

developer expectations

Developers have expectations about what language constructs mean and how implementations will process them. At the very least developers expect a translator to accept their existing source code and generate to a program image from it, the execution time behavior being effectively the same as the last implementation they used. Implementation vendors want to meet developer expectations whenever possible; it reduces the support overhead and makes for happier customers. Authors of translators spend a lot of time discussing what their customers expect of their product; however, detailed surveys of customer requirements are rarely carried out. What is available is existing source code. It is this existing code base that is often taken as representing developers expectations (translators should handle it without complaint, creating programs that deliver the expected behavior).

Three commonly encountered expectations are good performance, low code expansion ratio, and no surprising behavior; the following describes these expectations in more detail.

1. *C has a reputation for efficiency.* It is possible to write programs that come close to making optimum usage of processor resources. Writing such code manually relies on knowledge of the processor and how the translator used maps constructs to machine code. Very few developers know enough about these subjects to be able to consistently write very efficient programs. Your author sometimes has trouble predicting the machine code that would be generated when using the compilers he had written. As a general rule, your author finds it safe to say that any ideas developers have about the most efficient construct to use, at the statement level, are wrong. A cost effective solution is to not worry about statement level efficiency issues and let the translator look after things.
2. *C has a reputation for compactness.* The ratio of machine code instructions per C statement is often a small number compared to other languages. It could be said that C is a WYSIWYG language, the mapping from C statement to machine code being simple and obvious (leaving aside what an optimizer might subsequently do). This expectation was used by some members of WG14 as an argument against allowing the equality operator to have operands with structure type; a single operator potentially causing a large amount of code, a comparison for each member, to be generated. The introduction of the **inline** function-specifier has undermined this expectation to some degree (depending on whether **inline** is thought of as a replacement for function-like macros, or the inlining of functions that would not have been implemented as macros).

function specifier 1522
syntax
macro 1933
function-like

3. *C has a reputation for being a consistent language.* Developers can usually predict the behavior of the code they write. There are few dark corners whose accidental usage can cause constructs to behave in unexpected ways. While the C committee can never guarantee that there would never be any surprising behaviors, it did invest effort in trying to ensure that the least-surprising behaviors occurred.

4.2 The language specification

common implementations language specification

The C Standard does not specify everything that an implementation of it has to do. Neither does it prevent vendors from adding their own extensions. C is not a registered trademark that is policed to ensure implementations follow its requirements; unlike Ada, which until recently was a registered trademark, owned by the US Department of Defense, which required that an implementation pass a formal validation procedure before allowing it to be called Ada. The C language also has a history— it existed for 13 years before a formally recognized standard was ratified.

The commercial environments in which C was originally used have had some influence on its specification. The C language started life on comparatively small platforms and the source code of a translator (`pcc`, the portable C compiler[673]) was available for less than the cost of writing a new one. Smaller hardware vendors without an established customer base, were keen to promote portability of applications to their platform. Thus, there were very few widely accepted extensions to the base language. In this environment vendors tended to compete more in the area of available library functions. For this reason, significant developer communities, using different dialects of C, were not created. Established hardware vendors are not averse to adding language extensions specific to their platforms, which resulted in several widely used dialects of both Cobol and Fortran.

Implementation vendors have found that they can provide a product that simply follows the requirements contained in the C Standard. While some vendors have supplied options to support for some prestandard language features, the number of these features is small.

Although old source code is rarely rewritten, it still needs a host to run on. The replacement of old hosts by newer ones means that either existing source has to be ported, or new software acquired. In both cases it is likely that the use of prestandard C constructs will diminish. Many of the programs making use of C language dialects, so common in the 1980s, are now usually only seen executing on very old hosts. The few exceptions are discussed in the relevant sentences.

4.3 Implementation products

Translators are software products that have customers like any other application. The companies that produce them have shareholders to satisfy and, if they are to stay in business, need to take commercial issues into account. It has always been difficult to make money selling translators and the continuing improvement in the quality of Open Source C translators makes it even harder. Vendors who are still making most of their income by selling translators, as opposed to those who have to supply one as part of a larger sale, need to be very focused and tend to operate within specific markets.[1480] For instance, some choose to concentrate on the development process (speed of translation, integrated development environment, and sophisticated debugging tools), others on the performance of the generated machine code (Kuck & Associates, purchased by Intel, for parallelizing scientific and engineering applications, Code Play for games developers targeting the Intel x86 processor family). There are even specialists within niches. For instance, within the embedded systems market Byte Craft concentrates on translators for 8-bit processors. Vendors who are still making most of their income from selling other products (e.g., hardware or operating systems) sometimes include a translator as a loss leader. Given its size there is relatively little profit for Microsoft in selling a C/C++ translator; having a translator gives the company greater control over its significantly more profitable products (written in those languages) and, more importantly, mind-share of developers producing products for its operating systems.

It is possible to purchase a license for a C translator front-end from several companies. While writing one from scratch is not a significant undertaking (a few person years), writing anything other than a straightforward code generator can require a large investment. By their very nature, many optimization techniques deal with special cases, looking to fine-tune the use of processor resources. Ensuring that correct code is

generated, for all the myriad different combinations of events that can occur, is very time-consuming and expensive.

The performance of generated machine code is rarely the primary factor in developer selection of which translator to purchase, if more than one is available to choose from. Factors such as implicit Vendor preference (it is said that nobody is sacked for buying Microsoft), preference for the development environment provided, possessing existing code that is known to work well with a particular vendor's product, and many other possible issues. For this reason optimization techniques often take many years to find their way from published papers to commercial products, if at all.[1175]

Companies whose primary business is the sale of translators do not seem to grow beyond a certain point. The largest tend to have a turnover in the tens of millions of dollars. The importance of translators to companies in other lines of business has often led to these companies acquiring translator vendors, both for the expertise of their staff and for their products. Several database companies have acquired translator vendors to use their expertise and technology in improving the performance of the database products (the translators subsequently being dropped as stand-alone products).

Overall application performance is often an issue in the workstation market. Here vendors, such as HP, SGI, and IBM, have found it worthwhile investing in translator technology that improves the quality of generated code for their processors. Potential customers evaluating platforms using benchmarks will be looking at numbers that are affected by both processor and translator performance— the money to be made from multiple hardware sales being significantly greater than that from licensing a translator to relatively few developers. These companies consider it worthwhile to have an in-house translator development group.

GCC

GCC, the GNU C compiler[1278] (now renamed the GNU Compiler Collection; the term *gcc* will be used here to refer to the C compiler), was distributed in source code form long before Linux and the rise of the Open Source movement. Its development has been checkered, but it continues to grow from strength to strength. This translator was designed to be easily retargeted to a variety of different processors. Several processor vendors have provided, or funded ports of the back end to their products. Over time the optimizations performed by GCC have grown more sophisticated. This has a lot to do with researchers using GCC as the translator on which to implement and test their optimization ideas. On those platforms where its generated machine code does not rank first in performance, it usually ranks second.

The source code to several other C translators has also been released under some form of public use license. These include: `lcc`[448] along with `vpo` (very portable optimizer[113]), the SGIPRO C compiler[1237] (which performs many significant optimizations), the TenDRA C/C++ project,[44] Watcom,[1446] Extensible Interactive C (an interpreter),[156] and the Trimaran compiler system.[46]

The lesson to be drawn from these commercial realities is that developers should not expect a highly competitive market in language translators.[1480] Investing large amounts of money in translator development is unlikely to be recouped purely from sales of translators (some vendors make the investment to boost the sales of their processors). Developers need to work with what they are given.

4.4 Translation technology

translation technology

Translators for C exist within a community of researchers (interested in translation techniques) and also translators for other languages. Some techniques have become generally accepted as the way some construct is best implemented; some are dictated by trends that come and go. This book does not aim to document every implementation technique, but it may discuss the following.

- How implementations commonly map constructs for execution by processors.
- Unusual processor characteristics, which affect implementations.
- Common extensions in this area.
- Possible trade-offs involved in implementing a construct.
- The impact of common processor architectures on the C language.

In the early days of translation technology vendors had to invest a lot of effort simply to get them to run within the memory constraints of the available development environments. Many existed as a collection of separate programs, each writing output to be read by the succeeding phase, the last phase being assembler code that needed to be processed by an assembler.

Ever since the first Fortran translator[69] the quality of machine code produced has been compared to handwritten assembler. Initially translators were only asked to not produce code that was significantly worse than handwritten assembler; the advantages of not having to retrain developers (in new assembly languages) and rewrite applications outweigh the penalties of less performance. The fact that processors changed frequently, but software did not, was a constant reminder of the advantages of using a machine-independent language. Whether most developers stopped making the comparison against handwritten assembler because fewer of them knew any assembler, or because translators simply got better is an open issue. In some application domains the quality of code produced by translators is nowhere near that of handwritten assembler[1254] and many developers still need to write in machine code to be able to create usable applications.

Much of the early work on translators was primarily concerned with different language constructs and parsing them. A lot of research was done on various techniques for parsing grammars and tools for compressing their associated data tables. The work done at Carnegie Mellon on the PQCC project[847] introduced many of the ideas commonly used today. By the time C came along there were some generally accepted principles about how a translator should be structured.

A C translator usually operates in several phases. The first phase (called the *front-end* by compiler writers and often the parser by developers) performs syntax and semantic analysis of the source code and builds a tree representation (usually based on the abstract syntax); it may also map operations to an intermediate form (some translators have multiple intermediate forms, which get progressively lower as constructs proceed through the translation process) that has a lower-level representation than the source code but a higher-level than machine code. The last phase (often called the *back-end* by compiler writers or the *code generator* by developers) takes what is often a high-level abstract machine code (an intermediate code) and maps it to machine code (it may generate assembler or go directly to object code). Operations, such as storage layout and optimizations on the intermediate code, could be part of one of these phases, or be a separate phase (sometimes called the *middle-end* by compiler writers).

120 footnote 5

1354 storage layout

The advantage of generating machine code from intermediate code is a reduction in the cost of retargeting the translator to a new processor; the front-end remains virtually the same and it is often possible to reuse substantial parts of later passes. It becomes cost effective for a vendor to offer a translator that can generate machine code for different processors from the same source code. Many translators have a single intermediate code. GCC currently has one, called RTL (register transfer language), but may soon have more (a high-level, machine-independent, RTL, which is then mapped to a more machine specific form of RTL). Automatically deriving code generators from processor descriptions[209] sounds very attractive. However, until recently new processors were not introduced sufficiently often to make it cost effective to remove the human compiler written from the process. The cost of creating new processors, with special purpose instruction sets, is being reduced to the point where custom processors are likely to become very common and automatic derivation of code generators is essential to keep these costs down.[810, 843]

The other advantage of breaking the translator into several components is that it offers a solution to the problem caused by a common host limitation. Many early processors limited the amount of memory available to a program (64 K was a common restriction). Splitting a translator into independent components (the preprocessor was usually split off from the syntax and semantics processing as a separate program) enabled each of them to occupy this limited memory in turn. Today most translators have many megabytes of storage available to them; however, many continue to have internal structures designed when storage limitations were an important issue.

There are often many different ways of translating C source into machine code. Developers invariably want their programs to execute as quickly as possible and have been sold on the idea of translators that

perform code optimization. There is no commonly agreed on specification for exactly what a translator needs to do to be classified as optimizing, although claims made in a suitably glossy brochure is often sufficient for many developers.

4.4.1 Translator optimizations

translator optimizations

Traditionally optimizations have been aimed at reducing the time needed to execute a program (this is what the term *increasing program performance* is usually intended to mean) or reducing the size of the program image (this usually means the amount of storage occupied during program execution— consisting of machine code instructions, some literal values, and object storage). Many optimizations have the effect of increasing performance and reducing size. However, there are some optimizations that involve making a trade-off between performance and size.

The growth in mobile phones and other hand-held devices containing some form of processor have created a new optimization requirement— power minimization. Software developers want to minimize the amount of electrical power required to execute a program. This optimization requirement is likely to be new to readers; for this reason a little more detail is given at the end of this subsection.

Some of the issues associated with generating optimal machine code for various constructs are discussed within the sentences for those constructs. In some cases transformations are performed on a relatively high-level representation and are relatively processor-independent (see Bacon, Graham, and Sharp[70] for a review). Once the high-level representation is mapped to something closer to machine code, the optimizations can become very dependent on the characteristics of the target processor (Bonk and Rüde[136] look at number crunchers). The general techniques used to perform optimizations at different levels of representation can be found in various books.[9, 448, 524, 976]

The problems associated with simply getting a translator written became tractable during the 1970s. Since then the issues associated with translators have been the engineering problem of being able to process existing source code and the technical problem of generating high-quality machine code. The focus of code optimization research continues to evolve. It started out concentrating on expressions, then basic blocks, then complete functions and now complete programs. Hardware characteristics have not stood still either. Generating optimized machine code can now require knowledge of code and data cache behaviors, speculative execution, dependencies between instructions and their operands. There is also the issue of processor vendors introducing a range of products, all supporting the same instruction set but at different price levels and different internal performance enhancements; optimal instruction selection can now vary significantly across a single processor family.

Sometimes all the information about some of the components used by a program will not be known until it is installed on the particular host that executes it; for instance, any additional instructions supported over those provided in the base instruction set for that processor, the relative timings of instructions for that processor model, and the version of any dynamic linked libraries. These can also change because of other systems software updates. Also spending a lot of time during application installation generating an optimal executable program is not always acceptable to end users. One solution is to perform optimizations on the program while it is executing. Because most of the execution time usually occurs within a small percentage of a program's machine code, an optimizer only needs to concentrate on these areas. Experimental systems are starting to deliver interesting results.[742]

Thorup[1352] has shown that a linear (in the number of nodes and vertices in the control flow graph) algorithm for register allocation exists that is within a factor of seven (six if no short-circuit evaluation is used) of the optimal solution for any C program that does not contain `gotos`.

One way of finding optimal instruction sequences is to generate all possible sequences and to select the optimal one that provides the desired input to output transformation. Massalin[901] built a *superoptimizer* to do just that; it worked off-line and was not intended to be used to generate instruction sequences during translation. Bansal and Aiken[91] built a superoptimizer that is intended to be used within a translator to find optimal instruction sequences. The tools used various strategies to reduce the search space, e.g., pruning instruction sequences known to be nonoptimal and maintaining a database of previously generated optimal

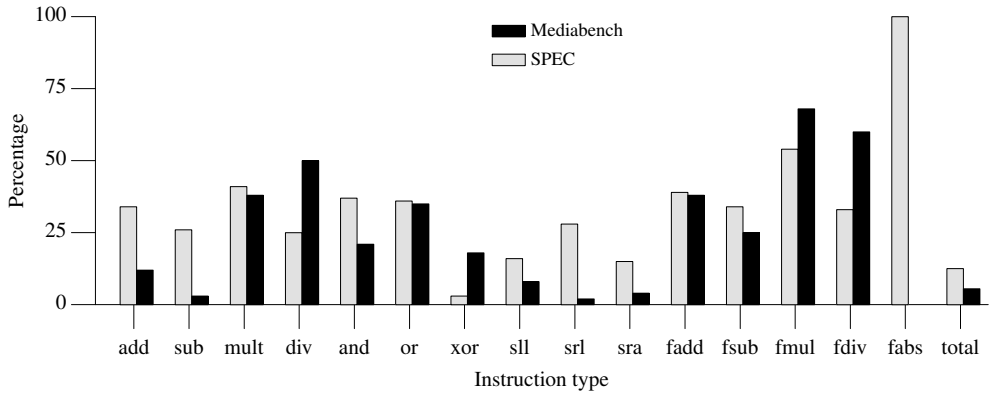


Figure 0.3: Dynamic frequency, percentage calculated over shown instructions (last column gives percentage of these instruction relative to all instructions executed) during execution of the SPEC and MediaBench benchmarks of some computational oriented instructions. Adapted from Yi and Lilja.[1492]

sequences.

Code optimization is a, translation time, resource-hungry process. To reduce the quantity of analysis that needs to be performed, optimizers have started to use information on a program's runtime characteristics. This profile information enables optimizers to concentrate resources on frequently executed sections of code (it also provides information on the most frequent control flow path in conditional statements, enabling the surrounding code to be tuned to this most likely case).[527, 1490] However, the use of profile information does not always guarantee better performance.[807]

The stability of execution profiles, that is the likelihood that a particular data set will always highlight the same sections of a program as being frequently executed is an important issue. A study by Chilimbi[227] found that data reference profiles, important for storage optimization, were stable, while some other researchers have found that programs exhibit different behaviors during different parts of their execution.[1227]

Optimizers are not always able to detect all possible savings. A study by Yi and Lilja[1492] traced the values of instruction operands during program execution. They found that a significant number of operations could have been optimized (see Figure 0.3) had one of their operand values been known at translation time (e.g., adding/subtracting zero, multiplying by 1, subtracting/dividing two equal values, or dividing by a power of 2).

Power consumption

optimize power consumption

The following discussion is based one that can be found in Hsu, Kremer and Hsiao.[602] The dominant source of power consumption in digital CMOS circuits (the fabrication technology used in mass-produced processors) is the dynamic power dissipation, P, which is based on three factors:

$$P \propto CV^2F \tag{0.1}$$

where C is the effective switching capacitance, V the supply voltage, and F the clock speed. A number of technical issues prevent the voltage from being arbitrarily reduced, but there are no restrictions on reducing the clock speed (although some chips have problems running at too low a rate).

For cpu bound programs simply reducing the clock speed does not usually lead to any significant saving in total power consumption. A reduction in clock speed often leads to a decrease in performance and the program takes longer to execute. The product of dynamic power consumption and time taken to execute remains almost unchanged (because of the linear relationship between dynamic power consumption and clock speed). However, random access memory is clocked at a rate that can be an order of magnitude less than the processor clock rate.

For memory-intensive applications a processor can be spending most of its time doing nothing but waiting for the results of load instructions to appear in registers. In these cases a reduction in processor clock rate will have little impact on the performance of a program. Program execution time, T, can be written as:

$$T = T_{cpu\_busy} + T_{memory\_busy} + T_{cpu\_and\_mem\_busy} \tag{0.2}$$

An analysis (using a processor simulation) of the characteristics of the following code:

```
1  for (j = 0; j < n; j++)
2     for (i = 0; i < n; i++)
3        accu += A[i][j];
```

found that (without any optimization) the percentage of time spent in the various subsystems was: *cpu_busy=0.01%*, *memory_busy=93.99%*, *cpu_and_mem_busy=6.00%*.

Given these performance characteristics, a factor of 10 reduction in the clock rate and a voltage reduction from 1.65 to 0.90 would reduce power consumption by a factor of 3, while only slowing the program down by 1% (these values are based on the Crusoe TM5400 processor).

Performing optimizations changes the memory access characteristics of the loop, as well as potentially reducing the amount of time a program takes to execute. Some optimizations and their effect on the performance of the preceding code fragment include the following:

loop control 1774
variable array 994
row-major storage order

- Reversing the order of the loop control variables (arrays in C are stored in row-major order) creates spatial locality, and values are more likely to have been preloaded into the cache: *cpu_busy=18.93%, memory_busy=73.66%, cpu_and_mem_busy=7.41%*

loop unrolling 1774
basic block 1710

- Loop unrolling increases the amount of work done per loop iteration (decreasing loop housekeeping overhead and potentially increasing the number of instructions in a basic block): *cpu_busy=0.67%, memory_busy=65.60%, cpu_and_mem_busy=33.73%*
- Prefetching data can also be a worthwhile optimization:[1408] *cpu_busy=0.67%, memory_busy=74.04%, cpu_and_mem_busy=25.29%*

These ideas are still at the research stage[601] and have yet to appear in commercially available translators (support, in the form of an instruction to change frequency/voltage, also needs to be provided by processor vendors).

At the lowest level processors are built from transistors, which are grouped together to form logic gates. In CMOS circuits power is dissipated in a gate when its output changes (i.e., it goes from 0 to 1, or from 1 to 0). Vendors interested in low power consumption try to minimize the number of gate transitions made during the operation of a processor. Translators can also help here. Machine code instructions consist of sequences of zeros and ones. Differences in bit patterns between adjacent instructions, encountered during program execution, cause gate transitions. The Hamming distance between two binary values (instructions) is the number of places at which their bit settings differ. Ordering instructions to minimize the total Hamming distance over the entire sequence will reduce power consumption in the instruction decoding area of a processor. Simulations based on such a reordering have shown savings of 13% to 20%.[824]

5 Execution environment

environment 104
execution

Two kinds of execution environment are specified in the C Standard, hosted and freestanding. These tend to affect implementations in terms of the quantity of resources provided (functionality to support library requirements— e.g., I/O, memory capacity, etc.).

There are classes of applications that tend to occur in only one of these environments, which can make it difficult to classify an issue as being application- or environment-based.

For hosted environments C programs may need to coexist with programs written in a variety of languages. Vendors often define a set of conventions that programs need to follow; for instance, how parameters are passed. The popularity of C for systems development means that such conventions are often expressed in C terms and the implementations of other languages have to adapt to the C view of how things work.

Existing environments have affected the requirements in the C Standard library. Unlike some languages the C language has tried to take the likely availability of functionality in different environments into account. For instance, the inability of some hosts to support signals has meant that there is no requirement that any signal handling (other than function stubs) be provided by an implementation. Minimizing the dependency on constructs being supported by a host environment enables C to be implemented on a wide variety of platforms. This wide implementability comes at the cost of some variability in supported constructs.

5.1 Host processor characteristics

host processors introduction

It is often recommended that developers ignore the details of host processor characteristics. However, the C language was, and continues to be, designed for efficient mapping to commonly available processors. Many of the benchmarks by which processor performance is measured are written in C. A detailed analysis of C needs to include a discussion of processor characteristics.

0 SPEC benchmarks

Many developers continue to show a strong interest in having their programs execute as quickly as possible, and write code that they think will achieve this goal. Developer interest in processor characteristics is often driven by this interest in performance and efficiency. Developer interest in performance could be considered to be part of the culture of programming. It does not seem to be C specific, although this language's reputation for efficiency seems to exacerbate it. There is sometimes a customer-driven requirement for programs to execute within resource constraints (execution time and memory being the most common constrained resources). In these cases detailed knowledge of processor characteristics may help developers tune an application (although algorithmic tuning invariably yields higher returns on investment). However, the information given in this book is at the level of a general overview. Developers will need to read processor vendor's manuals, very carefully, before they can hope to take advantage of processor-specific characteristics by changing how they write source code.

The following are the investment issues, from the software development point of view, associated with processor characteristics:

- Making effective use of processor characteristics usually requires a great deal of effort (for an in-depth tutorial on getting the best out of a particular processor see,[1343] for an example of performance forecasting aimed at future processors see Armstrong and Eigenmann[57]). The return on investment of this effort is often small (if not zero). Experience shows that few developers invest the time needed to systematically learn about individual processor characteristics. Preferring, instead, to rely on what they already know, articles in magazines, and discussions with other developers. A small amount of misguided investment is no more cost effective than overly excessive knowledgeable investment.
- Processors change more frequently than existing code. Although there are some application domains where it appears that the processor architecture is relatively fixed (e.g., the Intel x86 and IBM 360/370/3080/3090/etc.), the performance characteristics of different members of the same family can still vary dramatically. Within the other domains new processor architectures are still being regularly introduced. The likelihood of a change of processor remains an important issue.
- The commercial availability of translators capable of producing machine code, the performance of which is comparable to that of handwritten assembler (this is not true in some domains;[1254] one study[1395] found that in many cases translator generated machine code was a factor of 5–8 times slower than hand crafted assembler) means that any additional return on developer resource investment is likely to be low.

translator performance vs. assembler

Commercial and application considerations have caused hardware vendors to produce processors aimed at several different markets. It can be said that there are often family characteristics of processors within a

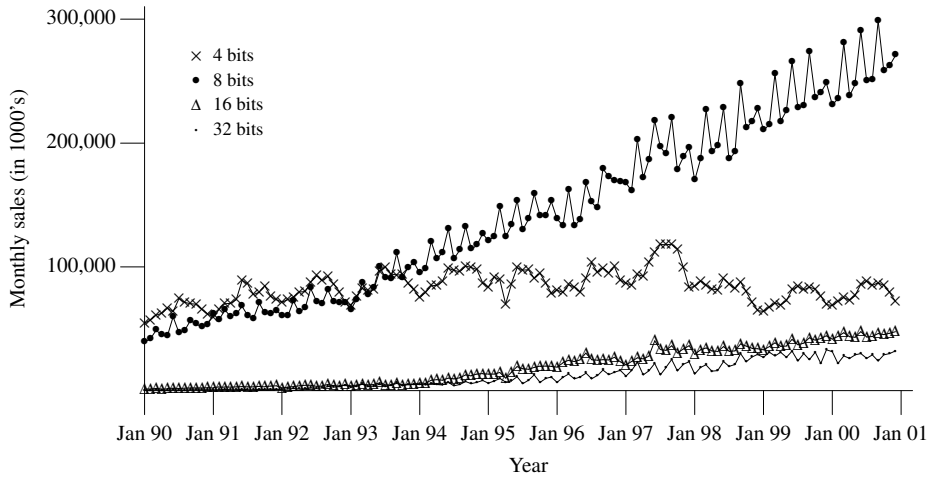


Figure 0.4: Monthly unit sales of microprocessors having a given bus width. Adapted from Turley[1369] (using data supplied by Turley).

given market, although the boundaries are blurred at times. It is not just the applications that are executed on certain kinds of processors. Often translator vendors target their products at specific kinds of processors. For instance, a translator vendor may establish itself within the embedded systems market. The processor architectures can have a dramatic effect on the kinds of problems that machine code generators and optimizers need to concern themselves with. Sometimes the relative performance of programs written in C, compared to handwritten assembler, can be low enough to question the use of C at all.

- *General purpose processors.* These are intended to be capable of running a wide range of applications. The processor is a significant, but not dominant, cost in the complete computing platform. The growing importance of multimedia applications has led many vendors to extend existing architectures to include instructions that would have previously only been found in DSP processors.[1254] The market size can vary from tens of millions (e.g., Intel x86[1335]) to hundreds of millions (e.g., ARM[1335]).
- *Embedded processors.* These are used in situations where the cost of the processor and its supporting chip set needs to be minimized. Processor costs can be reduced by reducing chip pin-out (which reduces the width of the data bus) and by reducing the number of transistors used to build the processor. The consequences of these cost savings are that instructions are often implemented using slower techniques and there may not be any performance enhancers such as branch prediction or caches (or even multiple and divide instructions, which have to be emulated in software). Some vendors offer a range of different processors, others a range of options within a single family, using the same instruction set (i.e., the price of an Intel i960 can vary by an order of magnitude, along with significant differentiation in its performance, packaging, and level of integration). The total market size is measured in billions of processors per year (see Figure 0.4).

DSP processors

- *Digital Signal Processors (DSP).* As the name suggests, these processors are designed for manipulating digital signals— for instance, decoding MPEG data streams, sending/receiving data via phone lines, and digital filtering types of applications. These processors are specialized to perform this particular kind of application very well; it is not intended that nondigital signal-processing applications ever execute on them. Traditionally DSPs have been used in applications where dataflow is the dominating factor;[120] making the provision of handcrafted library routines crucial. Recently new markets, such as telecoms and the automobile industry have started to use DSPs in a big way, and their applications have tended to be dominated by control flow, reducing the importance of libraries. Araújo[331] contains an up-to-date discussion on generating machine code for DSPs. The total worldwide market in 1999 was 0.6 billion processors;[1335] individual vendors expect to sell hundreds of millions of units.

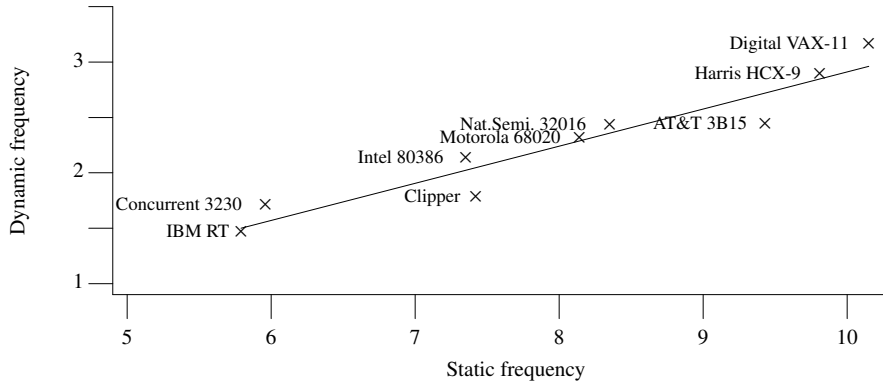


Figure 0.5: Dynamic/static frequency of *call* instructions. Adapted from Davidson.[325]

- *Application Specific Instruction-set Processors (ASIP).* Note that the acronym ASIC is often heard, this refers to an Application Specific Integrated Circuit— a chip that may or may not contain an instruction-set processor. These processors are designed to execute a specific program. The general architecture of the processor is fixed, but the systems developer gets to make some of the performance/resource usage (transistors) trade-off decisions. These decisions can involve selecting the word length, number of registers, and selecting between various possible instructions.[488] The cost of retargeting a translator to such program-specific ASIPs has to be very low to make it worthwhile. Processor description driven code generators are starting to appear,[844] which take the description used to specify the processor characteristics and build a translator for it. While the market for ASICs exceeds $10 billion a year, the ASIP market is relatively small (but growing).
- *Number crunchers.* The quest for ever-more performance has led to a variety of designs that attempt to spread the load over more than one processor. Technical problems associated with finding sufficient work, in existing source code (which tends to have a serial rather than parallel form) to spread over more than one processor has limited the commercial viability of such designs. They have only proven cost effective in certain, application-specific domains where the computations have a natural mapping to multiple processors. The cost of the processor is often a significant percentage of the complete computing device. The market is small and the customers are likely to be individually known to the vendor.[45] The use of clusters of low-price processors, as used in Beowulf, could see the demise of processors specifically designed for this market.[109]

There are differences in processor characteristics within the domains just described. Processor design evolves over time and different vendors make different choices about the best way to use available resources (on chip transistors). For a detailed analysis of the issues involved for the Sun UltraSPARC processor, see.[1498]

instruction profile for different processors

The profile of the kinds of instructions generated for different processors can differ in both their static and their dynamic characteristics, even within the same domain. This was shown quite dramatically by Davidson, Rabung, and Whalley[325] who measured static and dynamic instruction frequencies for nine different processors using the same translator (generating code for the different processors) on the same source files (see Figure 0.5). For a comparison of RISC processor instruction counts, based on the SPEC benchmarks, see McMahan and Lee.[921]

The following are the lessons to be learned from the later discussions on processor details:

- Source code that makes the best use of one particular processor is unlikely to make the best use of any other processor.
- Making the best use of a particular processor requires knowledge of how it works and measurements of the program running on it. Without the feedback provided by the measurement of dynamic program

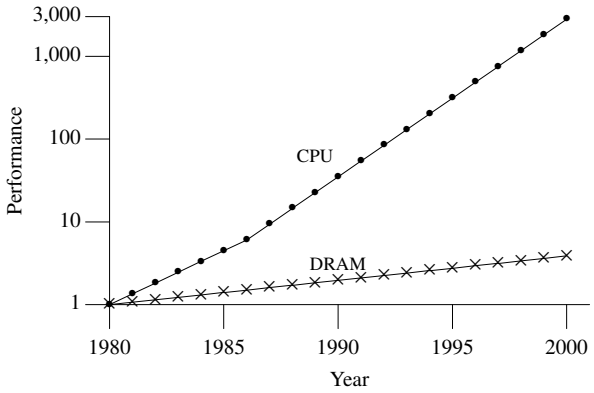


Figure 0.6: Relative performance of CPU against storage (DRAM), 1980==1. Adapted from Hennessy.[560]

behavior, it is almost impossible to tune a program to any host.

5.1.1 Overcoming performance bottlenecks

There continues to be a market for processors that execute programs more quickly than those currently available. There is a commercial incentive to build higher-performance processors. Processor design has reached the stage where simply increasing the processor clock rate does not increase rate of program execution.[1178] A processor contains a variety of units, any one of which could be the bottleneck that prevents other units from delivering full performance. Some of these bottlenecks, and their solutions, can have implications at the source code level (less than perfect branch predictions[695]) and others don't (the possibility of there being insufficient pins to support the data bandwidth required; pin count has only been increasing at 16% per year[181]).

Data and instructions have to be delivered to the processor, from storage, to keep up with the rate it handles them. Using faster memory chips to keep up with the faster processors is not usually cost effective. Figure 0.6 shows how processor performance has outstripped that of DRAM (the most common kind of storage used). See Dietz and Mattox[355] for measurements of access times to elements of arrays of various sizes, for 13 different Intel x86 compatible processors whose clock rates ranged from 100 MHz to 1700 MHz.

A detailed analysis by Agarwal, Hrishikesh, Keckler, and Burger[5] found that delays caused by the time taken for signals to travel through on-chip wires (12–32 cycles to travel the length of a chip using 35nm CMOS technology, clocked at 10GHz), rather than transistor switching speed, was likely to be a major performance factor in future processors. Various methods have been proposed[990] to get around this problem, but until such processor designs become available in commercially significant quantities they are not considered further here.

cache

Cache

A commonly used solution to the significant performance difference between a processor and its storage is to place a small amount of faster storage, a *cache*, between them. Caching works because of locality of reference. Having accessed storage location X, a program is very likely to access a location close to X in the very near future. Research has shown[560] that even with a relatively small cache (i.e., a few thousand bytes) it is possible to obtain significant reductions in accesses to main storage.

Modern, performance-based processors have two or more caches. A level 1 cache (called the *L1 cache*), which can respond within a few clock cycles (two on the Pentium 4, four on the UltraSPARC III), but is relatively small (8 K on the Pentium 4, 64 K on the UltraSPARC III), and a level 2 cache (called the *L2 cache*) which is larger but not as quick (256 K/7 clocks on the Pentium 4). Only a few processors have further levels of cache. Main storage is significantly larger, but its contents are likely to be more than 250 clock cycles away.

Developer optimization of memory access performance is simplest when targeting processors that contain a cache, because the hardware handles most of the details. However, there are still cases where developers may need to manually tune memory access performance (e.g., application domains where large, sophisticated hardware caches are too expensive, or where customers are willing to pay for their applications to execute as fast as possible on their existing equipment). Cache behavior when a processor is executing more than one program at the same time can be quite complex.[219, 1358]

The locality of reference used by a cache applies to both instructions and data. To maximize locality of reference, translators need to organize instructions in the order that an executing program is most likely to need them and allocate object storage so that accesses to them always fill the cache with values that will be needed next. Knowing which machine code sequences are most frequently executed requires execution profiling information on a program. Obtaining this information requires effort by the developer. It is necessary to instrument and execute a program on a representative set of data. This data, along with the original source is used by some translators to create a program image having a better locality of reference. It is also possible to be translator-independent by profiling and reorganizing the basic blocks contained in executable programs. Tomiyama and Yasuura[1354] used linear programming to optimize the layout of basic blocks in storage and significantly increased the instruction cache hit rate. Running as a separate pass after translation also reduces the need for interactive response times; the analysis took more than 10 hours on a 85 MHz microSPARC-II.

Is the use of a cache by the host processor something that developers need to take into account? Although every effort has been made by processor vendors to maximize cache performance and translator vendors are starting to provide the option to automatically tune the generated code based on profiling information,[547] sometimes manual changes to the source (by developers) can make a significant difference. It is important to remember that any changes made to the source may only make any significant changes for one particular processor implementation. Other implementations within a processor family may share the same instruction set but they could have different cache behaviors. Cache-related performance issues are even starting to make it into the undergraduate teaching curriculum.[816]

A comparison by Bahar, Calder, and Grunwald[79] showed that code placement by a translator could improve performance more than a hardware-only solution; the two combined can do even better. In some cases the optimizations performed by a translator can affect cache behavior, for instance loop unrolling. Translators that perform such optimizations are starting to become commercially available.[252] The importance of techniques for tuning specific kinds of applications are starting to be recognized (transaction processing as in Figure 0.8,[11] numerical computations[1367]).

1774 loop unrolling

Specific cases of how optimizers attempt to maximize the benefits provided by a processor's cache are discussed in the relevant C sentences. In practice these tend to be reorganizations of the sequence of instructions executed, not reorganizations of the data structures used. Intel[628] provides an example of how reorganization of a data structure can improve performance on the Pentium 4:

```
1  struct {
2          float x, y, z,   r, g, b;
3          } a_screen_3D[1000];
4  struct {
5          float x[1000], y[1000], z[1000];
6          float r[1000], g[1000], b[1000];
7          } b_screen_3D;
8  struct {
9          float x[4], y[4], z[4];
10         float r[4], g[4], b[4];
11         } c_screen_3D[250];
```

The structure declaration used for `a_screen_3D` might seem the obvious choice. However, it is likely that operations will involve either the tuple `x`, `y`, and `z`, or the tuple `r`, `g`, and `b`. A cache line on the Pentium 4 is 64 bytes wide, so a fetch of one of the `x` elements will cause the corresponding `r`, `g`, and `b` elements to be loaded. This is a waste of resource usage if they are not accessed. It is likely that all elements of the

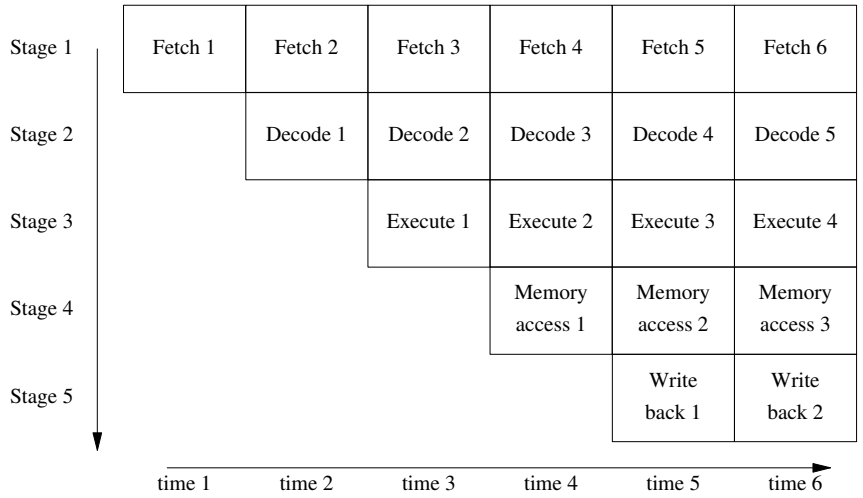


Figure 0.7: Simplified diagram of some typical stages in a processor instruction pipeline: Instruction fetch, decode, execute, memory access, and write back.

array will be accessed in sequence and the structure declaration used for `b_screen_3D` makes use of this algorithmic information. An access to an element of `x` will cause subsequent elements to be loaded into the cache line, ready for the next iteration. The structure declaration, suggested by Intel, for `c_screen_3D` makes use of a Pentium 4 specific characteristic; reading/writing 16 bytes from/to 16-byte aligned storage is the most efficient way to use the storage pipeline. Intel points to a possible 10% to 20% performance improvement through modifications that optimize cache usage; a sufficiently large improvement to warrant using the nonobvious, possibly more complex, data structures in some competitive markets.

storage dividing up

Dividing up storage

Many host operating systems provide the ability for programs to make use of more storage than the host has physical memory (so-called *virtual memory*). This virtual memory is divided up into units called *pages*, which can be *swapped* out of memory to disk when it is not needed.[560] There is a severe performance penalty on accesses to data that has been swapped out to disk (i.e., some other page needs to be swapped out and the page holding the required data items swapped back into memory from disk). Developers can organize data accesses to try to minimize this penalty. Having translators do this automatically, or even having them insert code into the program image to perform the swapping at points that are known to be not time-critical is a simpler solution.[973]

Speeding up instruction execution

processor pipeline

A variety of techniques are used to increase the number of instructions executed per second. Most processors are capable of executing more than one instruction at the same time. The most common technique, and one that can affect program behavior, is instruction *pipelining*. Pipelining breaks instruction execution down into a series of stages (see Figure 0.7). Having a different instruction processed by each stage at the same time does not change the execution time of a single instruction. But it does increase the overall rate of instruction execution because an instruction can complete at the end of every processor cycle. Many processors break down the stages shown in Figure 0.7 even further. For instance, the Intel Pentium 4 has a 20-stage pipeline.

signal interrupt 191
abstract machine processing

The presence of a pipeline can affect program execution, depending on processor behavior when an exception is raised during instruction execution. A discussion on this issue is given elsewhere.

Other techniques for increasing the number of instructions executed per second include: *VLIW* (Very Long Instruction Word) in which multiple operations are encoded in one long instruction, and parallel execution in which a processor contains multiple instruction execution units.[1342] These techniques have no more direct

impact on program behavior than instruction pipelining. In practice translators have had difficulty finding long sequences of instructions that can be executed in some concurrent fashion. Some help (e.g., source code annotations) from the developer is still needed for these processors to approach peak performance.

5.2 Runtime library

An implementation's runtime library handles those parts of a program that are not directly translated to machine code. Calls to the functions contained in this library may occur in the source or be generated by a translator (i.e., to some internal routine to handle arithmetic operations on values of type **`long long`**). The runtime library may be able to perform the operation completely (e.g., the trigonometric functions) or may need to call other functions provided by the host environment (e.g., O/S function, not C implementation functions).

6 Measuring implementations

Measuring implementations

Although any number of different properties of an implementation might be measured, the most commonly measured is execution time performance of the generated program image. In an attempt to limit the number of factors influencing the results, various organizations have created sets of test programs— *benchmarks*— that are generally accepted within their domain. Some of these test programs are discussed below (SPEC, the Transaction Processing council, Embedded systems, Linpack, and DSPSTONE). In some application areas the size of the program image can be important, but there are no generally accepted benchmarks for comparing size of program image. The growth in sales of mobile phones and other hand-held devices has significantly increased the importance of minimizing the electrical energy consumed by a program (the energy consumption needs of different programs performing the same function are starting to be compared[103]).

141 program image

A good benchmark will both mimic the characteristics of the applications it is intended to be representative of, and be large enough so that vendors cannot tune their products to perform well on it without also performing well on the real applications. The extent to which the existing benchmarks reflect realistic application usage is open to debate. Not only can different benchmarks give different results, but the same benchmark can exhibit different behavior with different input.[378] Whatever their shortcomings may be the existing benchmarks are considered to be the best available (they are used in almost all published research).

It has long been an accepted truism that programs spend most of their time within loops and in particular a small number of such loops. Traditionally most processor-intensive applications, that were commercially important, have been scientific or engineering based. A third kind of application domain has now become commercially more important (in terms of hardware vendors making sales)— data-oriented applications such as transaction processing and data mining.

1763 iteration statement syntax

Some data-oriented applications share a characteristic with scientific and engineering applications in that a large proportion of their time is spent executing a small percentage of the code. However, it has been found that for Online Transaction Processing (OLTP), specifically the TPC-B benchmarks, the situation is more complicated.[1143] Recent measurements of four commercial databases running on an Intel Pentium processor showed that the processor spends 60% of its time stalled[11] (see Figure 0.8).

0 TPC-B

A distinction needs to be made between characteristics that are perceived, by developers, to make a difference and those that actually do make a difference to the behavior of a program. Discussion within these Common Implementation sections is concerned with constructs that have been shown, by measurement, to make a difference to the execution time behavior of a program. Characteristics that relate to perceived differences fall within the realm of discussions that occur in the Coding guideline sections.

The measurements given in the Common Implementation sections tend to be derived from the characteristics of a program while it is being executed— dynamic measurements. The measurements given in the Usage sections tend to be based on what appears in the source code— static measurements.

6.1 SPEC benchmarks

SPEC benchmarks

Processor performance based on the SPEC (Standard Performance Evaluation Corporation, `http://www.spec.org`) benchmarks are frequently quoted by processor and implementation vendors. Academic research on optimizers often base their performance evaluations on the programs in the SPEC suite. SPEC benchmarks

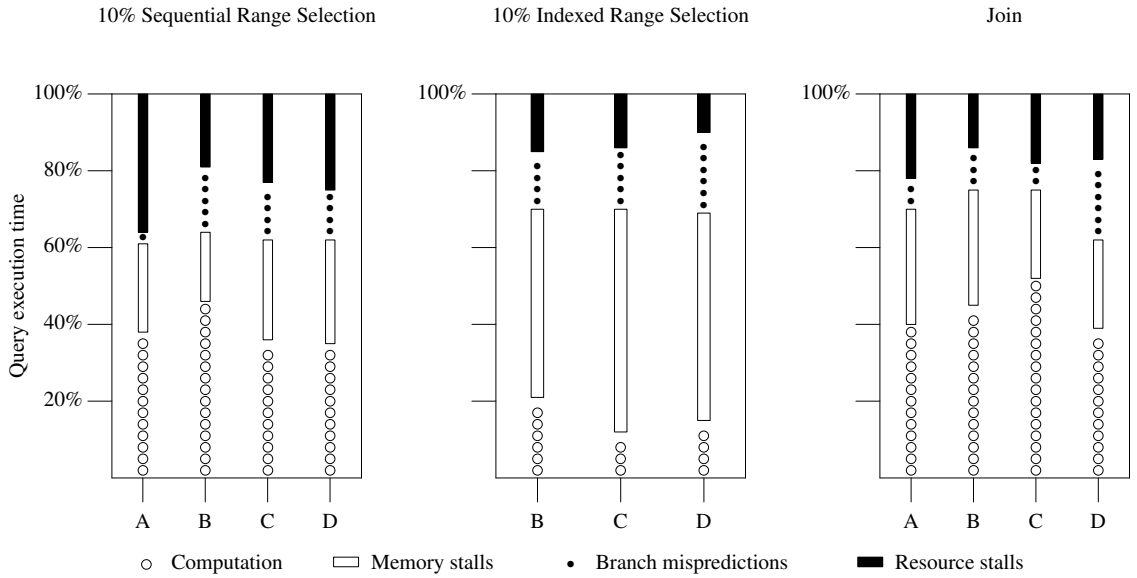


Figure 0.8: Execution time breakdown, by four processor components (bottom of graphs) for three different application queries (top of graphs). Adapted from Ailamaki.[11]

cover a wide range of performance evaluations: graphics, NFS, mailservers, and CPU.[365] The CPU benchmarks are the ones frequently used for processor and translator measurements.

The SPEC CPU benchmarks are broken down into two groups, the integer and the floating-point programs; these benchmarks have been revised over the years, the major releases being in 1989, 1992, 1995, and 2000. A particular set of programs is usually denoted by combining the names of these components. For instance, SPECint95 is the 1995 integer SPEC benchmark and SPECfp2000 is the 2000 floating-point benchmark.

The SPEC CPU benchmarks are based on publicly available source code (written in C for the integer benchmarks and, predominantly, Fortran and C for the floating-point). The names of the programs are known and versions of the source code are available on the Internet. The actual source code used by SPEC may differ slightly because of the need to be able to build and execute identical programs on a wide range of platforms (any changes needed to a program's source to enable it to be built are agreed to by the SPEC membership).

A study by Saavedra and Smith[1190] investigated correlations between constructs appearing in the source code and execution time performance of benchmarks that included SPEC.

6.2 Other benchmarks

benchmarks

The SPEC CPU benchmarks had their origins in the Unix market. As such they were and continue to be aimed at desktop and workstation platforms. Other benchmarks that are often encountered, and the rationale used in their design, include the following:

- DSPSTONE[1395] is a DSP-oriented set of benchmarks,
- The characteristics of programs written for embedded applications are very different.[390] The EDN Embedded Microprocessor Benchmarking Consortium (EEMBC, pronounced Embassy — http://www.eembc.org), was formed in 1997 to develop standard performance benchmarks for the embedded market (e.g., telecommunications, automotive, networking, consumer, and office equipment). They currently have over 40 members and their benchmark results are starting to become known.
- MediaBench[818] is a set of benchmarks targeted at a particular kind of embedded application—multimedia and communications. It includes programs that process data in various formats, including JPEG, MPEG, GSM, and postscript.

- The Olden benchmark[198] attempts to measure the performance of architectures based on a distributed memory. Olden benchmark
- The Stanford ParalleL Applications for SHared memory (SPLASH, now in its second release as SPLASH-2[14481]), is a suite of parallel applications intended to facilitate the study of centralized and distributed shared-address-space multiprocessors.
- The TPC-B benchmark from the Transaction Processing Performance Council (TPC). TPC-B

> *TPC-B models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, which includes updating the balance of the branch the customer belongs to and the teller from which the transaction is submitted. It also adds an entry to the history table which keeps a record of all submitted transactions.* Ranganathan[1143]

6.3 Processor measurements

Processor vendors also measure the characteristics of executing programs. Their reason is to gain insights that will enable them to build better products, either faster versions of existing processors or new processors. What are these measurements based on? The instructions executed by a processor are generated by translators, which may or may not be doing their best with the source they are presented with. Translator vendors may, or may not, have tuned their output to target processors with known characteristics. Fortunately this book does not need to concern itself further with this problem.

Processor measurements have been used to compare different processors,[248] predict how many instructions a processor might be able to issue at the same time,[1259] and tune arithmetic operations.[885] Processor vendors are not limited to using benchmarks or having access to source code to obtain useful information; Lee[820] measured the instruction characteristics of several well-known Windows NT applications.

Coding Guidelines

7 Introduction

The intent of these coding guidelines is to help management minimize the cost of ownership of the source code they are responsible for. The guidelines take the form of prepackaged recommendations of which source constructs to use, or not use, when more than one option is available. These coding guidelines sit at the bottom layer of what is potentially a complex, integrated software development environment. coding guidelines introduction

Adhering to a coding guideline is an immediate cost. The discussion in these coding guidelines' sections is intended to help ensure that this cost payment is a wise investment yielding savings later.

The discussion in this section provides the background material for what appears in other coding guideline sections. It is also the longest section of the book and considers the following:

- The financial aspects of software development and getting the most out of any investment in adhering to coding guidelines.
- Selecting, adhering to, and deviating from guidelines.
- Applications and their influence on the source that needs to be written.
- Developers; bounding the limits, biases, and idiosyncrasies of their performance.

There are other Coding guideline subsections containing lengthy discussions. Whenever possible such discussions have been integrated into the C sentence-based structure of this book (i.e., they occur in the relevant C sentences).

The term used in this book to describe people whose jobs involve writing source code is *software developer*. The term *programmer* tends to be associated with somebody whose only job function is to write software. A

typist might spend almost 100% of the day typing. People do not spend all their time directly working on source code (in most studies, the time measured on this activity rarely rises above 25%), therefore the term programmer is not appropriate. The term software developer, usually shortened to *developer*, was chosen because it is relatively neutral, but is suggestive of somebody whose primary job function involves working with source code.

Developers often object to following coding guidelines, which are often viewed as restricting creative freedom, or forcing them to write code in some unnatural way. Creative freedom is not something that should be required at the source code implementation level. While particular ways of doing things may appeal to individual developers, such usage can be counter-productive. The cost to the original developer may be small, but the cost to subsequent developers (through requiring more effort by them to work with code written that way) may not be so small.

8 Source code cost drivers

coding guidelines cost drivers

Having source code occupy disk space rarely costs very much. The cost of ownership for source code is incurred when it is used. Possible uses of source code include:

- modifications to accommodate customer requests which can include fixing faults;
- major updates to create new versions of a product; and
- ports to new platforms, which can include new versions of platforms already supported.

These coding guideline subsections are applicable during initial implementation and subsequent modifications at the source code level. They do not get involved in software design issues, to the extent that these are programming language-independent. The following are the underlying factors behind these cost drivers:

- *Developer characteristics (human factors).* Developers fail to deduce the behavior of source code constructs, either through ignorance of C or because of the limits in human information processing (e.g., poor memory of previously read code, perception problems leading to identifiers being misread, or information overload in short-term memory) causing faults to be introduced. These issues are dealt with here in the Coding guideline subsections.
- *Translator characteristics.* A change of translator can result in a change of behavior. Changes can include using a later version of the translator originally used, or a translator from a different vendor. Standards are rarely set in stone and the C Standard is certainly not. Variations in implementation behavior permitted by the standard means that the same source code can produce different results. Even the same translator can have its behavior altered by setting different options, or by a newer release. Differences in translator behavior are discussed in Commentary and Common Implementations subsections. Portability to C++ and C90 translators is dealt with in their respective sections.
- *Host characteristics.* Just like translator behavior this can vary between releases (updates to system libraries) and host vendors. The differences usually impact the behavior of library calls, not the language. These issues are dealt with in Common Implementation sections.
- *Application characteristics.* Programs vary in the extent to which they need to concern themselves with the host on which they execute— for instance, accessing memory ports. They can also place different demands on language constructs— for instance, floating-point or dynamic memory allocation. These issues are dealt with under Usage, indirectly under Common Implementations and here in Coding Guideline sections.

Usage 0 1

- *Product testing.* The complexity of source code can influence the number of test cases that need to be written and executed. This complexity can be affected by design, algorithmic and source code construct selection issues. The latter can be the subject of coding guidelines.

coding 0 guidelines testability

Covering all possible source code issues is impossible. Frequency of occurrence is used to provide a cutoff filter. The main purpose of the information in the Usage sections is to help provide evidence for what filtering to apply.

Usage 0 1

8.1 Guideline cost/benefit

When a guideline is first encountered it is educational. It teaches developers about a specific problem that others have encountered and that they are likely to encounter. This is a one-time learning cost (that developers are likely to have to pay at some time in their careers). People do forget, so there may be a relearning cost. (These oversights are the sort of thing picked up by an automated guideline enforcement tool, jogging the developer's memory in the process.)

coding guidelines importance

Adhering to guidelines requires an investment in the form of developer's time. Like all investments it needs to be made on the basis that a later benefit will provide an adequate return. It is important to bear in mind that failure to recoup the original investment is not the worst that can happen. The value of lost opportunity through being late to market with a product can equal the entire development budget. It is management's responsibility to select those coding guidelines that have a return on investment applicable to a particular project.

0 NPV

A set of guidelines can be viewed as a list of recommended coding practices, the economic cost/benefit of which has been precalculated and found to be acceptable. This precalculation, ideally, removes the need for developers to invest in performing their own calculations. (Even in many situations where they are not worthwhile, the cost of performing the analysis is greater than the cost of following the guideline.)

Researchers[90, 1321] are only just starting to attempt to formally investigate the trade-off involved between the cost of creating maintainable software and the cost of maintaining software.

A study by Visaggio[1423] performed a retrospective analysis of a reengineering process that had been applied to a legacy system containing 1.5 M lines. The following is his stated aim:

Visaggio[1423]

1. Guidelines are provided for calculating the quality and economic scores for each component; These can be reused in other projects, although they can and must also be continually refined with use;

2. A model for determining the thresholds on each axis is defined; the model depends on the quality and economics policy adopted by the organization intending to renew the legacy system;

3. A decision process is included, that helps to establish which renewal process should be carried out for each component; this process may differ for components belonging to the same quadrant and depends on the targets the organization intends to attain with the renewal process.

8.1.1 What is the cost?

Guidelines may be more or less costly to follow (in terms of modifying, or not using, constructs once their lack of conformance to a guideline is known). Estimating any cost change caused by having to use constructs not prohibited by a guideline will vary from case to case. It is recognized that the costs of following a guideline recommendation can be very high in some cases. One solution is the deviation mechanism, which is discussed elsewhere.

coding guidelines the cost

0 deviations coding guidelines

Guidelines may be more or less easy to flag reliably from a static analysis tool point of view. The quality of static analysis tools is something that developers need to evaluate when making a purchase decision. These coding guidelines recognize the difficulties in automating some checks by indicating that some should be performed as part of code reviews.

0 code reviews

All guidelines are given equal weight in terms of the likelihood of not adhering to them causing a fault. Without data correlating a guideline not being followed to the probability of the containing code causing a fault, no other meaningful options are available.

8.1.2 What is the benefit?

What is the nature of the benefit obtained from an investment in adhering to coding guidelines? These coding guidelines assume that the intended final benefit is always financial. However, the investment proposal may not list financial benefits as the immediate reason for making it. Possible other reasons include:

coding guidelines the benefit

- mandated by some body (e.g., regulatory authority, customer Q/A department);

- legal reasons— companies want to show that they have used industry best practices, whatever they are, in the event of legal action being taken against them;
- a mechanism for controlling source code: The purpose of this control may be to reduce the dependency on a particular vendor's implementation (portability issues), or it may be an attempt to overcome inadequacies in developer training.

Preventing a fault from occurring is a benefit. How big is this benefit (i.e., what would the cost of the fault have been? How is the cost of a fault measured?) Is it in terms of the cost of the impact on the end user of experiencing the fault in the program, or is it the cost to the vendor of having to deal with it being uncovered by their customers (which may include fixing it)? Measuring the cost to the end user is very difficult to do, and it may involve questions that vendors would rather have left unasked. To simplify matters these guidelines are written from the point of view of the vendor of the product containing software. The cost we consider is the cost to fix the fault multiplied by the probability of the fault needing to be fixed (fault is found and customer requirements demand a fix).

8.1.3 Safer software?

coding guidelines
safer software

Coding guidelines, such as those given in this book, are often promoted as part of the package of measures to be used during the development of safety-critical software.

The fact that adherence to guideline recommendations may reduce the number of faults introduced into the source by developers is primarily an economic issue. The only difference between safety critical software and other kinds of software is the level of confidence required that a program will behave as intended. Achieving a higher level of confidence often involves a higher level of cost. While adherence to guideline recommendations may reduce costs and enable more confidence level boosting tasks to be performed, for the same total cost, management may instead choose to reduce costs and not perform any additional tasks.

Claiming that adhering to coding guidelines makes programs safer suggests that the acceptance criteria being used are not sufficient to achieve the desired level of confidence on their own (i.e., reliance is being placed on adherence to guideline recommendations reducing the probability of faults occurring in sections of code that have not been fully tested).

An often-heard argument is that some language constructs are the root cause of many faults in programs and that banning the use of these constructs leads to fewer faults. While banning the use of these constructs may prevent them from being the root cause of faults, there is rarely any proof that the alternative constructs used will not introduce as many faults, if not more, than the constructs they replace.

This book does not treat *safety-critical* as being a benefit of adherence to guideline recommendations in its own right.

8.2 Code development's place in the universe

development context

Coding guidelines need to take account of the environment in which they will be applied. There are a variety of reasons for creating programs. Making a profit is a common rationale and the only one considered by these coding guidelines. Writing programs for enjoyment, by individuals, involves reasons of a personal nature, which are not considered in this book.

A program is created by developers who will have a multitude of reasons for doing what they do. Training and motivating these developers to align there interests with that of the organization that employs them is outside the scope of this book, although staffing issues are discussed.

coding 0 guidelines staffing

Programs do not exist in isolation. While all applications will want fault-free software, the importance assigned to faults can depend on the relative importance of the software component of the total package. This relative importance will also influence the percentage of resources assigned to software development and the ability of the software manager to influence project time scales.

The kind of customers an organization sells to, can influence the software development process. There are situations where effectively there is a single customer. For instance, a large organization paying for the development of a bespoke application will invariably go through a formal requirements analysis, specification,

design, code, test, and handover procedure. Much of the research on software development practices has been funded by and for such development projects. Another example is software that is invisible to the end user, but is part of a larger product. Companies and projects differ as to whether software controls the hardware or vice versa (the hardware group then being the customer).

Most *Open Source* software development has a single customer, the author of the software.[483, 1010] In this case the procedures followed are likely to be completely different from those followed by paying customers. In a few cases Open Source projects involving many developers have flourished. Several studies[951] have investigated some of the group dynamics of such cooperative development (where the customer seems to be the members of a core team of developers working on the project). While the impact of this form of production on traditional economic structures is widely thought to be significant,[114] these guidelines still treat it as a form of production, which has different cost/benefit cost drivers; whether the motivating factors for individual developers are really any different is not discussed here.

When there are many customers, costs are recouped over many customers, who usually pay less than the development cost of the software. In a few cases premium prices can be charged by market leaders, or by offering substantial customer support. The process used for development is not normally visible to the customer. Development tends to be led by marketing and is rarely structured in any meaningful formal way; in fact too formal a process could actively get in the way of releasing new products in a timely fashion.

Research by Carmel[201] of 12 firms (five selling into the mass market, seven making narrow/direct sales) involved in packaged software development showed that the average firm has been in business for three years, employed 20 people, and had revenues of $1 million (1995 figures).

As pointed out by Carmel and others, time to market in a competitive environment can be crucial. Being first to market is often a significant advantage. A vendor that is first, even with a very poorly architected, internally, application often gets to prosper. While there may be costs to pay later, at least the company is still in business. A later market entrant may have a wonderfully architected product that has scope for future expansion and minimizes future maintenance costs, but without customers it has no future.

A fundamental problem facing software process improvement is how best to allocate limited resources, to obtain optimal results. Large-scale systems undergo continuous enhancement and subcontractors may be called in for periods of time. There are often relatively short release intervals and a fixed amount of resources. These characteristics prohibit revolutionary changes to a system. Improvements have to be made in an evolutionary fashion.

Coding guidelines need to be adaptable to these different development environments. Recognizing that guideline recommendations will be adapted, it is important that information on the interrelationship between them is made available to the manager. These interrelationships need to be taken into account when tailoring a set of guideline recommendations.

8.3 Staffing

The culture of information technology appears to be one of high staff turnover[961] (with reported annual turnover rates of 25% to 35% in Fortune 500 companies). coding guidelines staffing

If developers cannot be retained on a project new ones need to be recruited. There are generally more vacancies than there are qualified developers to fill them. Hiring staff who are less qualified, either in application-domain knowledge or programming skill, often occurs (either through a conscious decision process or because the developer's actual qualifications were not appreciated). The likely competence of future development staff may need to be factored into the acceptable complexity of source code.

A regular turnover of staff creates the need for software that does not require a large investment in upfront training costs. While developers do need to be familiar with the source they are to work on, companies want to minimize familiarization costs for new staff while maximizing their productivity. Source code level guideline recommendations can help reduce familiarization costs in several ways:

- Not using constructs whose behavior varies across translator implementations means that recruitment does not have to target developers with specific implementation experience, or to factor in the cost of

retraining— it will occur, usually through on-the-job learning.

- Minimizing source complexity helps reduce the cognitive effort required from developers trying to comprehend it.
- Increased source code memorability can reduce the number of times developers need to reread the same source.
- Visible source code that follows a consistent set of idioms can take advantage of people's natural ability to categorize and make deductions based on these categorizes.

Implementing a new project is seen, by developers, as being much more interesting and rewarding that maintaining existing software. It is common for the members of the original software to move on to other projects once the one they are working is initially completed. Studies by Couger and Colter[291] investigated various approaches to motivating developers working on maintenance activities. They identified the following two factors:

1. *The motivating potential of the job*, based on skill variety required, the degree to which the job requires completion as a whole (task identity), the impact of the job on others (task significance), degree of freedom in scheduling and performing the job, and feedback from the job (used to calculate a *Motivating Potential Score*, MPS).
2. *What they called an individual's growth need strength (GNS)*, based on a person's need for personal accomplishment, to be stimulated and challenged.

The research provided support for the claim that MPS and GNS could be measured and that jobs could be tailored, to some degree, to people. Management's role was to organize the work that needed to be done so as to balance the MPS of jobs against the GNS of the staff available.

It is your author's experience that very few companies use any formally verified method for measuring developer characteristics, or fitting their skills to the work that needs to be done. Project staffing is often based on nothing more than staff availability and a date by which the tasks must be completed.

8.3.1 Training new staff

developer training

Developers new to a project often need to spend a significant amount of time (often months) building up their knowledge base of a program's source code.[1239] One solution is a training program, complete with well-documented introductions, road maps of programs, and how they map to the application domain, all taught by well-trained teachers. While this investment is cost effective if large numbers of people are involved, most source code is worked on by a relatively small number of people. Also most applications evolve over time. Keeping the material up-to-date could be difficult and costly, if not completely impractical. In short, the cost exceeds the benefit.

In practice new staff have to learn directly from the source code. This may be supplemented by documentation, provided it is reasonably up-to-date. Other experienced developers who have worked on the source may also be available for consultation.

8.4 Return on investment

ROI

The risk of investing in the production of software is undertaken in the expectation of receiving a return that is larger than the investment. Economists have produced various models that provide an answer for the question: "What return should I expect from investing so much money at such and such risk over a period of time?"

Obtaining reliable estimates of the risk factors, the size of the financial investment, and the time required is known to be very difficult. Thankfully, they are outside the scope of this book. However, given the prevailing situation within most development groups, where nobody has had any systematic cost/benefit analysis training, an appreciation of the factors involved can provide some useful background.

Minimizing the total cost of a software product (e.g., balancing the initial development costs against subsequent maintenance costs) requires that its useful life be known. The risk factors introduced by third parties (e.g., competitive products may remove the need for continued development, customers may not purchase the product) mean that there is the possibility that any investment made during development will never be realized during maintenance because further work on the product never occurs.

The physical process of writing source code is considered to be so sufficiently unimportant that doubling the effort involved is likely to have a minor impact on development costs. This is the opposite case to how most developers view the writing process. It is not uncommon for developers to go to great lengths to reduce the effort needed during the writing process, paying little attention to subsequent effects of their actions; reports have even been published on the subject.[1125]

8.4.1 Some economics background

Before going on to discuss some of the economic aspects of coding guidelines, we need to cover some of the basic ideas used in economics calculations. The primary quantity that is used in this book is Net Present Value (NPV). NPV

8.4.1.1 Discounting for time

A dollar today is worth more than a dollar tomorrow. This is because today's dollar can be invested and start earning interest immediately. By tomorrow it will have increased in value. The present value (PV) of a future payoff, C, can be calculated from:

$$PV = discount\ factor \times C \tag{0.3}$$

where the $discountfactor$ is less than one. It is usually represented by:

$$discount\ factor = \frac{1}{1+r} \tag{0.4}$$

where r is known as the rate of return; representing the amount of reward demanded by investors for accepting a delayed payment. The rate of return is often called the *discount rate* or the *opportunity cost* of capital. It is often quoted over a period of a year, and the calculation for PV over n years becomes:

$$PV = \frac{C}{(1+r)^{\mathrm{n}}} \tag{0.5}$$

By expressing all future payoffs in terms of present value, it is possible to compare them on an equal footing.

Example (from Raffo[1136]). A manager has the choice of spending $250,000 on the purchase of a test tool, or the same amount of money on hiring testers. It is expected that the tool will make an immediate cost saving of $500,000 (by automating various test procedures). Hiring the testers will result in a saving of $750,000 in two years time. Which is the better investment (assuming a 10% discount rate)?

$$PV_{tool} = \frac{\$500,000}{(1+0.10)^0} = \$500,000 \tag{0.6}$$

$$PV_{testers} = \frac{\$750,000}{(1+0.10)^2} = \$619,835 \tag{0.7}$$

Based on these calculations, hiring the testers is the better option (has the greatest present value).

8.4.1.2 Taking risk into account

The previous example did not take risk into account. What if the tool did not perform as expected, what if some of the testers were not as productive as hoped? A more realistic calculation of present value needs to take the risk of future payoffs not occurring as expected into account.

A risky future payoff is not worth as much as a certain future payoff. The risk is factored into the discount rate to create an *effective discount rate*: $k = r + \theta$ (where r is the risk-free rate and θ a premium that depends on the amount of risk). The formulae for present value becomes:

$$PV = \frac{C}{1 + k^{\mathrm{n}}} \tag{0.8}$$

Recognizing that both r and θ can vary over time we get:

$$PV = \sum_{i=1}^{t} \frac{return_i}{1 + k_i} \tag{0.9}$$

where $return_i$ is the return during period i.

Example. Repeating the preceding example, but assuming a 15% risk premium for the testers option.

$$PV_{tool} = \frac{\$500,000}{(1 + 0.10)^0} = \$500,000 \tag{0.10}$$

$$PV_{testers} = \frac{\$750,000}{(1 + 0.10 + 0.15)^2} = \$480,000 \tag{0.11}$$

Taking this risk into account shows that buying the test tool is the better option.

8.4.1.3 Net Present Value

Future payoffs do not just occur, an investment needs to be made. A quantity called the *Net Present Value* (NPV) is generally considered to lead to the better investment decisions.[154] It is calculated as:

$$NPV = PV - investment\ cost \tag{0.12}$$

Example (from Raffo[1136]). A coding reading initiative is expected to cost $50,000 to implement. The expected payoff, in two years time, is $100,000. Assuming a discount rate of 10%, we get:

$$NPV = \frac{\$100,000}{1 \times 10^2} - \$50,000 = \$32,645 \tag{0.13}$$

Several alternatives to NPV, their advantages and disadvantages, are described in Chapter five of Brealey[154] and by Raffo.[1136] One commonly seen rule within rapidly changing environments is the payback rule. This requires that the investment costs of a project be recovered within a specified period. The payback period is the amount of time needed to recover investment costs. A shorter payback period being preferred to a longer one.

8.4.1.4 Estimating discount rate and risk

The formulae for calculating value are of no use unless reliable figures for the discount rate and the impact of risk are available. The discount rate represents the risk-free element and the closest thing to a risk-free investment is government bonds and securities. Information on these rates are freely available. Governments face something of a circularity problem in how they calculate the discount rate for their own investments. The US government discusses these issues in its *Guidelines and Discount Rates for Benefit-Cost Analysis of Federal Programs*[1456] and specifies a rate of 7%.

Analyzing risk is a much harder problem. Information on previous projects carried out within the company can offer some guidance on the likelihood of developers meeting productivity targets. In a broader context the market conditions also need to be taken into account, for instance: how likely is it that other companies will bring out competing products? Will demand for the application still be there once development is complete?

One way of handling these software development risks is for companies to treat these activities in the same way that options are used to control the risk in a portfolio of stocks. Some very sophisticated models and formula for balancing the risks involved in holding a range of assets (e.g., stocks) have been developed. The match is not perfect in that these methods rely on a liquid market, something that is difficult to achieve using people (moving them to a new project requires time for them to become productive). A number of researchers[412, 1313, 1478] have started to analyze some of the ways these methods might be applied to creating and using options within the software development process.

8.5 Reusing software

It is rare for a single program to handle all the requirements of a complete application. An application is often made up of multiple programs and generally there is a high degree of similarity in many of the requirements for these programs. In other cases there may be variations in a hardware/software product. Writing code tailored to each program or product combination is expensive. Reusing versions of the same code in multiple programs sounds attractive.

In practice code reuse is a complex issue. How to identify the components that might be reusable, how much effort should be invested in writing the original source to make it easy to reuse, how costs and benefits should be apportioned are a few of the questions.

A survey of the economic issues involved in software reuse is provided by Wiles.[1464] These coding guidelines indirectly address code reuse in that they recommend against the use of constructs that can vary between translator implementations.

8.6 Using another language

A solution that is sometimes proposed to get around problems in C that are seen as the root cause of many faults is to use another language. Commonly proposed languages include Pascal, Ada, and recently Java. These languages are claimed to have characteristics, such as strong typing, that help catch faults early and reduce maintenance costs.

Ada using

In 1987 the US Department of Defense mandated Ada (DoD Directive 3405.1) as the language in which bespoke applications, written for it, had to be written. The aim was to make major cost savings over the full lifetime of a project (implementation and maintenance, throughout its operational life); the higher costs of using Ada during implementation[1155] being recovered through reduced maintenance costs over its working lifetime.[1504] However, a crucial consideration had been overlooked in the original cost analysis. Many projects are canceled before they become operational.[404, 1281] If the costs of all projects, canceled or operational, are taken into account, Ada is not the most cost-effective option. The additional cost incurred during development of projects that are canceled exceeds the savings made on projects that become operational. The directive mandating the use of Ada was canceled in 1997.[1394]

Proposals to use other languages sometimes have more obvious flaws in their arguments. An analysis of why Lisp should be used[411] is based on how that language overcomes some of the C-inherent problems, while overlooking its own more substantial weaknesses (rather like proposing that people hop on one leg as a solution to wearing out two shoes by walking on two).

The inability to think through a reasoned argument, where choice of programming language is concerned, is not limited to academic papers[994] (5.3.11 Safe Subsets of Programming languages).

The use of software in applications where there is the possibility of loss of life, or serious injury, is sometimes covered by regulations. These regulations often tend to be about process— making sure that various checks are carried out. But sometimes subsets of the C language have been defined (sometimes called by the name safe subsets). The associated coding guideline is that constructs outside this subset not be used. Proof for claiming that use of these subsets result in safer programs is nonexistent. The benefit of following coding guidelines is discussed elsewhere.

coding 0 guidelines the benefit

8.7 Testability

coding guidelines testability

This subsection is to provide some background on testing programs. The purpose of testing is to achieve a measurable degree of confidence that a program will behave as expected. Beizer[107] provides a practical introduction to testing.

Testing is often written about as if its purpose were to find faults in applications. Many authors quote figures for the cost of finding a fault, looking for cost-effective ways of finding them. This outlook can lead to an incorrectly structured development process. For instance, a perfect application will have an infinite cost per fault found, while a very badly written application will have a very low cost per fault found. Other figures often quoted involve the cost of finding faults in different phases of the development process. In particular, the fact that the cost per fault is higher, the later in the process it is discovered. This observation about relative costs often occurs purely because of how development costs are accounted for. On a significant development effort equipment and overhead costs tend to be fixed, and there is often a preassigned number of people working on a particular development phase. These costs are not likely to vary by much, whether there is a single fault found or 100 faults. However, it is likely that there will be significantly fewer faults found in later phases because most of them will have been located in earlier phases. Given the fixed costs that cannot be decreased, and the smaller number of faults, it is inevitable that the cost per fault will be higher in later phases.

Many of the faults that exist in source code are never encountered by users of an application. Examples of such faults are provided in a study by Chou, Yang, Chelf, Hallem, and Engler[232] who investigated the history of faults in the Linux kernel (found using a variety of static analysis tools). The source of different releases of the Linux kernel is publicly available (for this analysis 21 snapshots of the source over a seven year period were used). The results showed how faults remained in successive releases of code that was used for production work in thousands (if not hundreds of thousands) of computers. The average fault lifetime, before being fixed, or the code containing it ceasing to exist was 1.8 years.

The following three events need to occur for a fault to become an application failure:

1. A program needs to execute the statement containing the fault.
2. The result of that execution needs to infect the subsequent data values, another part of the program.
3. The infected data values must propagate to the output.

The probability of a particular fault affecting the output of an application for a given input can be found by multiplying together the probability of the preceding three events occurring for that set of input values. The following example is taken from Voas:[1427]

```
1  #include <math.h>
2  #include <stdio.h>
3
4  void quadratic_root(int a, int b, int c)
5  /*
6   * If one exists print one integral solution of:
7   *  ax^2 + bx + c = 0
8   */
9  {
```

```
10  int d,
11      x;
12
13  if (a != 0)
14     {
15     d = (b * b) - (5 * a * c); /* Fault, should multiply by 4. */
16     if (d < 0)
17        x = 0;
18     else
19        x = (sqrt(d) / (2 * a)) - b;
20     }
21  else
22     x = -(c / b);
23
24  if ((a * x * x + b * x + c) == 0)
25     printf("%d is an integral solution\n", x);
26  else
27     printf("There is no integral solution\n");
28  }
```

Execution of the function `quadratic_root` has four possibilities:

1. The fault is not executed (e.g., `quadratic_root(0, 3, 6)`).
2. The fault is executed but does not infect any of the data (e.g., `quadratic_root(3, 2, 0)`).
3. The fault is executed and the data is infected, but it does not affect the output (e.g., `quadratic_root(1, -1, -12)`).
4. The fault is executed and the infected data causes the output to be incorrect (e.g., `quadratic_root(10, 0, 10)`).

This program illustrates the often-seen situations of a program behaving as expected because the input values used were not sufficient to turn a fault in the source code into an application failure during program execution.

Testing by execution examines the source code in a different way than is addressed by these coding guidelines. One looks at only those parts of the program (in translated form) through which flow of control passes and applies specific values, the other examines source code in symbolic form.

A study by Adams[3] looked at faults found in applications over time. The results showed (see Table 0.1) that approximately one third of all detected faults occurred on average every 5,000 years of execution time. Only around 2% of faults occurred every five years of execution time.

Table 0.1: Percentage of reported problems having a given mean time to first problem occurrence (in months, summed over all installations of a product) for various products (numbered 1 to 9), e.g., 28.8% of the reported faults in product 1 were, on average, first reported after 19,000 months of program execution time (another 34.2% of problems were first reported after 60,000 months). From Adams.[3]

Product	19	60	190	600	1,900	6,000	19,000	60,000
1	0.7	1.2	2.1	5.0	10.3	17.8	28.8	34.2
2	0.7	1.5	3.2	4.5	9.7	18.2	28.0	34.3
3	0.4	1.4	2.8	6.5	8.7	18.0	28.5	33.7
4	0.1	0.3	2.0	4.4	11.9	18.7	28.5	34.2
5	0.7	1.4	2.9	4.4	9.4	18.4	28.5	34.2
6	0.3	0.8	2.1	5.0	11.5	20.1	28.2	32.0
7	0.6	1.4	2.7	4.5	9.9	18.5	28.5	34.0
8	1.1	1.4	2.7	6.5	11.1	18.4	27.1	31.9
9	0.0	0.5	1.9	5.6	12.8	20.4	27.6	31.2

8.8 Software metrics

metrics introduction

In a variety of engineering disciplines, it is possible to predict, to within a degree of uncertainty, various behaviors and properties of components by measuring certain parameters and matching these measurements against known behaviors of previous components having similar measurements. A number of software metrics (software measurements does not sound as scientific) are based on the number of lines of source code. Comments are usually excluded from this count. What constitutes a line is at times fiercely debated.[1054] The most commonly used count is based on a simple line count, ignoring issues such as multiple statements on one line or statements spanning more than one line.

The results from measurements of software are an essential basis for any theoretical analysis.[419] However, some of the questions people are trying to answer with measurements of source code have serious flaws in their justification. Two commonly asked questions are the effort needed to implement a program (before it is implemented) and the number of faults in a program (before it is shipped to customers). Fenton attempted to introduce a degree of rigour into the use of metrics.[417,418]

COCOMO

The COCOMO project (COnstructive COst Model, the latest release is known as COCOMO II) is a research effort attempting to produce an Open Source, public domain, software cost, effort, and schedule for developing new software development. Off-the-shelf, untuned models have been up to 600% inaccurate in their estimates. After Bayesian tuning models that are within 30% of the actual figures 71% of the time have been built.[235] Effort estimation is not the subject of this book and is not discussed further.

These attempts to find meaningful measures all have a common goal — the desire to predict. However, most existing metrics are based on regression analysis models, they are not causal models. To build these models, a number of factors believed to affect the final result are selected, and a regression analysis is performed to calculate a correlation between them and the final results. Models built in this way will depend on the data from which they were built and the factors chosen to correlate against. Unlike a causal model (which predicts results based on "telling the story",[417]) there is no underlying theory that explains how these factors interact. For a detailed critique of existing attempts at program defect prediction based on measures of source code and fault history, see Fenton.[417]

The one factor that existing fault-prediction models ignore is the human brain/mind. The discussion in subsequent sections should convince the reader that source code complexity only exists in the mind of the reader. Without taking into account the properties in the reader's mind, it is not possible to calculate a complexity value. For instance, one frequently referenced metric is Halstead's software science metric, which uses the idea of the *volume* of a function. This *volume* is calculated by counting the operators and operands appearing in that function. There is no attempt to differentiate functions containing a few complex expressions from functions containing many simple expressions; provided the total and unique operand/operator count is the same, they will be assigned the same complexity.

9 Background to these coding guidelines

coding guidelines background to

These coding guidelines are conventional, if a little longer than most, in the sense that they contain the usual exhortation not to use a construct, to do things a particular way, or to watch out for certain problems. They are unconventional because of the following:

- An attempt has been made to consider the impact of a prohibition— do the alternatives have worse cost/benefit?
- Deviations are suggested— experience has shown that requiring a yes/no decision on following a guideline recommendation can result in that recommendation being ignored completely. Suggesting deviations can lead to an increase in guideline recommendations being followed by providing a safety valve for the awkward cases.
- Economics is the only consideration— it is sometimes claimed that following guideline recommendations imbues software with properties such as being better or safer. Your author does not know of any way of measuring betterness in software. The case for increased safety is discussed elsewhere.

coding guidelines the benefit 0

- An attempt has been made to base those guideline recommendations that relate to human factors on the experimental results and theories of cognitive psychology.

0 cognitive psychology

The wording used in these guideline recommendations is short and to the point (and hopefully unambiguous). It does assume some degree of technical knowledge. There are several ISO standards[635, 649] dealing with the wording used in the specification of a computer language. The principles of designing and documenting procedures to be carried out by others are thoroughly covered by Degani and Wiener.[336]

It is all very well giving guideline recommendations for developers to follow. But, how do they do their job. How were they selected? When do they apply? These are the issues discussed in the following sections.

9.1 Culture, knowledge, and behavior

culture of C

Every language has a culture associated with its use. A culture entails thinking about and doing certain things in a certain way.[1015] How and why these choices originally came about may provide some interesting historical context and might be discussed in other sections of this book, but they are generally not relevant to Coding guideline sections.

Culture is perhaps too grand a word for the common existing practices of C developers. Developers are overconfident and insular enough already without providing additional blankets to wrap themselves in. The term *existing practice* is both functional and reduces the possibility of aggrandizement.

Existing practices could be thought of as a set of assumptions and expectations about how things are done (in C). The term *C style* is sometimes used to describe these assumptions and expectations. However, this term has so many different meanings, for different developers, in different contexts, that its use is very prone to misunderstanding and argument. Therefore every effort will be made to stay away from the concept of style in this book.

0 coding guidelines
coding style

In many ways existing practice is a *meme machine*.[125] Developers read existing code, learn about the ideas it contains, and potentially use those ideas to write new code. Particular ways of writing code need not be useful to the program that contains them. They only need to appear to be useful to the developer who writes the code, or fit in with a developer's preferred way of doing things. In some cases developers do not thoroughly analyze what code to write, they follow the lead of others. Software development has its fads and fashions, just like any other information-driven endeavor.[124]

Before looking at the effect of existing practice on coding guidelines we ought to ask what constitutes existing practice. As far as the guideline recommendations in this book are concerned, what constitutes existing practice is documented in the Usage subsections. Developers are unlikely to approach this issue in such a scientific way. They will have worked in one or more application domains, been exposed to a variety of source code, and discussed C with a variety of other developers. While some companies might choose to tune their guidelines to the practices that apply to specific application domains and working environments, the guideline recommendations in this book attempt to be generally applicable.

0 measure-ments
5ESS

Existing practices are not always documented and, in some cases, developers cannot even state what they are. Experienced developers sometimes use expressions such as *the C way of doing things*, or *I feel*. When asked what is meant by these expressions, they are unable to provide a coherent answer. This kind of human behavior (knowing something without being able to state what it is) has been duplicated in the laboratory.

implicit learning

- A study by Lewicki, Hill and Bizot[849] demonstrated the effect of implicit learning on subjects expectations, even when performing a task that contained no overt learning component. In this study, while subjects watched a computer screen a letter was presented in one of four possible locations. Subjects had to press the button corresponding to the location of the letter as quickly as possible. The sequence of locations used followed a consistent, but complex, pattern. The results showed subjects' response times continually improving as they gained experience. The presentation was divided into 17 segments of 240 trials (a total of 4,080 letters), each segment was separated by a 10-second break. The pattern used to select the sequence of locations was changed after the 15th segment (subjects were not told about the existence of any patterns of behavior). When the pattern changed, the response times

immediately got worse. After completing the presentation subjects were interviewed to find out if they had been aware of any patterns in the presentation; they had not.

letter patterns
implicit learning

- A study by Reber and Kassin[1150] compared implicit and explicit pattern detection. Subjects were asked to memorize sets of words containing the letters *P*, *S*, *T*, *V*, or *X*. Most of these words had been generated using a finite state grammar. However, some of the sets contained words that had not been generated according to the rules of this grammar. One group of subjects thought they were taking part in a purely memory-based experiment; the other group was also told to memorize the words but was also told of the existence of a pattern to the letter sequences and that it would help them in the task if they could deduce this pattern. The performance of the group that had not been told about the presence of a pattern almost exactly mirrored that of the group who had been told on all sets of words (pattern words only, pattern plus non-pattern words, non-pattern words only). Without being told to do so, subjects had used patterns in the words to help perform the memorization task.
- A study carried out by Berry and Broadbent[117] asked subjects to perform a task requiring decision making using numerical quantities. In these experiments subjects were told that they were in charge of a sugar production factory. They had to control the rate of sugar production to ensure it kept at the target rate of 9,000 tons. The only method of control available to them was changing the size of the workforce. Subjects were not told anything about the relationship between the current production rate, the number of workers and previous production rates. The starting point was 600 workers and an output rate of 6,000 tons. Subjects had to specify the number of workers they wished to employ and were then told the new rate of production (interaction was via a terminal connected to a computer). At the end of the experiment, subjects had to answer a questionnaire about the task they had just performed. The results showed that although subjects had quickly learned to keep the rate of sugar production close to the desired level, they were unable to verbalize how they achieved this goal.

The studies performed by these and other researchers demonstrate that it is possible for people to perform quite complex tasks using knowledge that they are not consciously aware of having. By working with other C developers and reading existing C source code, developers obtain the nonverbalized knowledge that is part of the unwritten culture of C. This knowledge is expressed by developers having expectations and making assumptions about software development in C.

Another consequence of being immersed within existing practice is that developers use the characteristics of different entities to form categories. This categorization provides a mechanism for people to make generalizations based on relatively small data sets. A developer working with C source code which has been written by other people will slowly build up a set of assumptions and expectations of the general characteristics of this code.

A study by Nisbett, Krantz, Jepson and Kunda[1014] illustrates peoples propensity to generalize, based on past experience. Subjects were given the following scenario. (Some were told that three samples of each object was encountered, while other subjects were told that 20 samples of each object was encountered.)

> Imagine that you are an explorer who has landed on a little-known island in the Southeastern Pacific. You encounter several new animals, people, and objects. You observe the properties of your "samples" and you need to make guesses about how common these properties would be in other animals, people, or objects of the same type:
>
> 1. suppose you encounter a new bird, the shreeble. It is blue in color. What percent of all shreebles on the island do you expect to be blue?
> 2. suppose the shreeble you encounter is found to nest in a eucalyptus tree, a type of tree that is fairly common on this island. What percentage of all shreebles on the island do you expect to nest in a eucalyptus tree?

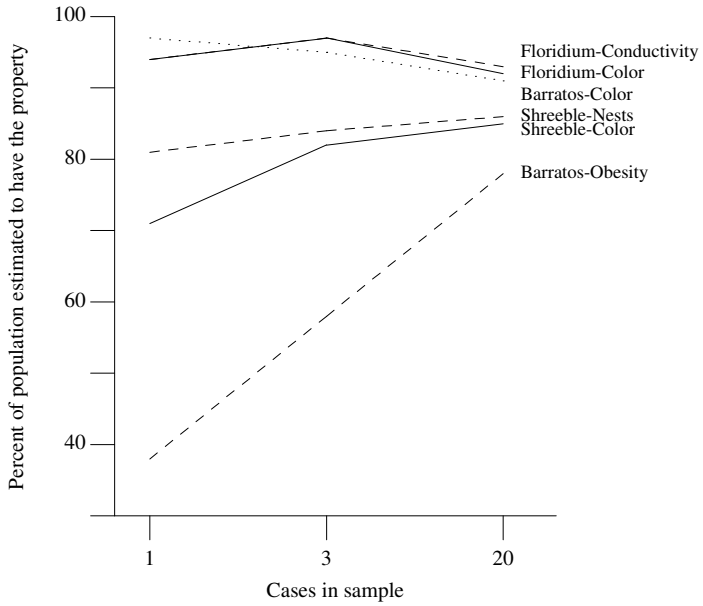


Figure 0.9: Percentage of population estimated to have the sample property against the number of cases in the sample. Adapted from Nisbett.[1014]

3. suppose you encounter a native, who is a member of a tribe called the Barratos. He is obese. What percentage of the male Barratos do you expect to be obese?
4. suppose the Barratos man is brown in color. What percentage of male Barratos do you expect be brown (as opposed to red, yellow, black, or white)?
5. suppose you encounter what the physicist on your expedition describes as an extremely rare element called floridium. Upon being heated to a very high temperature, it burns with a green flame. What percentage of all samples of floridium found on the island do you expect to burn with a green flame?
6. suppose the samples of floridium, when drawn into a filament, is found to conduct electricity. What percentage of all samples of floridium found on the island do you expect to conduct electricity?

The results show that subjects used their knowledge of the variability of properties in estimating the probability that an object would have that property. For instance, different samples of the same element are not expected to exhibit different properties, so the number of cases in a sample did not influence estimated probabilities. However, people are known to vary in their obesity, so the estimated probabilities were much lower for the single sample than the 20 case sample.

The lesson to be learned here is a general one concerning the functionality of objects and functions that are considered to form a category. Individual members of a category (e.g., a source file or structure type created by a developer) should have properties that would not be surprising to somebody who was only familiar with a subset of the members (see Figure 0.9).

Having expectations and making assumptions (or more technically, using inductive reasoning) can be useful in a slowly changing world (such as the one inhabited by our ancestors). They provide a framework from which small amounts of information can be used to infer, seemingly unconnected (to an outsider), conclusions. Is there a place for implicit expectations and assumptions in software development? A strong

case can be made for saying that any thought process that is not based on explicit knowledge (which can be stated) should not be used when writing software. In practice use of such knowledge, and inductive reasoning based on it, appears to play an integral role in human thought processes. A guideline recommendation that developers not use such thought processes may be difficult, if not impossible, to adhere to.

These coding guidelines don't seek to change what appears to be innate developer (human) behavior. The approach taken by these guidelines is to take account of the thought processes that developers use, and to work within them. If developers have expectations and make assumptions, then the way to deal with them is to find out what they are and to ensure that, where possible, source code follows them (or at least does not exhibit behavior that differs significantly from that expected). This approach means that these recommendations are tuned to the human way of comprehending C source code.

statement 1707
visual layout
declaration 1348
visual layout
identifier 792
syntax

The issue of implicit knowledge occurs in several coding guidelines.

9.1.1 Aims and motivation

developer motivations

What are developers trying to do when they read and write source code? They are attempting to satisfy a variety of goals. These goals can be explicit or implicit. One contribution cognitive psychology can make is to uncover the implicit goals, and perhaps to provide a way of understanding their effects (with the aim of creating guideline recommendations that minimize any undesirable consequences). Possible developer aims and motives include (roughly from higher level to lower level) the following:

- Performing their role in a development project (with an eye on promotion, for the pleasure of doing a good job, or doing a job that pays for other interests).
- Carrying out a program-modification task.
- Extracting information from the source by explicitly choosing what to pay attention to.

cogni-0
tive effort

- Minimizing cognitive effort; for instance, using heuristics rather than acquiring all the necessary information and using deductive logic.
- Maximizing the pleasure they get out of what they are doing.

belief main-0
tenance

- Belief maintenance: studies have found that people interpret evidence in ways that will maintain their existing beliefs.

The act of reading and writing software has an immediate personal cost. It is the cognitive load on a developer's brain (physical effort is assumed to be small enough that it has no significant cost, noticeable to the developer). Various studies have shown that people try to minimize cognitive effort when performing tasks.[458] A possible consequence of minimizing this effort is that people's actions are not always those that would be predicted on the basis of correct completion of the task at hand. In other words, people make mistakes because they do not invest sufficient effort to carry out a task correctly.

cost/accuracy trade-off

When attempting to solve a problem, a person's cognitive system is assumed to make cost/accuracy trade-offs. The details of how it forms an estimate of the value, cost, and risk associated with an action, and carries out the trade-off analysis is not known. A study by Fu and Gray[458] provides a good example of the effects of these trade-offs on the decisions made by people when performing a task. Subjects were given the task of copying a pattern of colored blocks (on a computer-generated display). To carry out the task subjects had to remember the color of the block to be copied and its position in the target pattern, a memory effort. A perceptual-motor effort was introduced by graying out the various areas of the display where the colored blocks were visible. These grayed out areas could be made temporarily visible using various combinations of keystrokes and mouse movements. When performing the task, subjects had the choice of expending memory effort (learning the locations of different colored blocks) or perceptual-motor effort (using keystrokes and mouse movements to uncover different areas of the display). A subject's total effort was equal to the sum of the perceptual motor effort and the memory storage and recall effort. The extremes of possible effort combinations are: (1) minimize the memory effort by remembering the color and position of a single block, which requires the perceptual-motor effort of uncovering the grayed out area for every block, or (2) minimize

perceptual effort by remembering information on as many blocks as possible (this requires uncovering fewer grayed areas).

The subjects were split into three groups. The experiment was arranged such that one group had to expend a low effort to uncover the grayed out areas, the second acted as a control, and the third had to expend a high effort to uncover the grayed out areas. The results showed that the subjects who had to expend a high perceptual-motor effort, uncovered grayed out area fewer times than the other two groups. These subjects also spent longer looking at the areas uncovered, and moved more colored blocks between uncoverings. The subjects faced with a high perceptual-motor effort reduced their total effort by investing in memory effort. Another consequence of this switch of effort investment, to use of memory, was an increase in errors made.

When reading source code, developers may be faced with the same kind of decision. Having looked at and invested effort in memorizing information about a section of source code, should they invest perceptual-motor effort when looking at a different section of source that is affected by the previously read source to verify the correctness of the information in their memory? A commonly encountered question is the C language type of an object. A developer has to decide between searching for the declaration or relying on information in memory.

A study by Schunn, Reder, Nhouyvanisvong, Richards, and Stroffolino[1206] found that a subject's degree of familiarity with a problem was a better predictor, than retrievability of an answer, of whether subjects would attempt to retrieve or calculate the answer to a problem.

The issue of cognitive effort vs. accuracy in decision making is also discussed elsewhere.

0 effort vs. accuracy decision making

Experience shows that many developers believe that code *efficiency* is an important attribute of code quality. This belief is not unique to the culture of C and has a long history.[772] While efficiency remains an issue in some application domains, these coding guidelines often treat efficiency as a cause of undesirable developer behavior that needs to be considered (with a view handling the possible consequences).

Experience has shown that some developers equate visual compactness of source code with runtime efficiency of the translated program. While there are some languages where such a correlation exists (e.g., some implementations of Basic, mostly interpreter based and seen in early hobbyist computers, perform just in time translation of the source code), it does not exist for C. This is an issue that needs to be covered during developer education.

visually compact code efficiency belief

Experience has also shown that when presented with a choice developer decisions are affected by their own estimates of the amount of typing they will need to perform. Typing minimization behavior can include choosing abbreviated identifier names, using cut-and-paste to copy sections of code, using keyboard short-cuts, and creating editor macros (which can sometimes require significantly more effort than they save).

typing minimization

9.2 Selecting guideline recommendations

No attempt has been made to keep the number of guideline recommendations within a prescribed limit. It is not expected that developers should memorize them. Managers are expected to select guidelines based on their cost effectiveness for particular projects.

guideline recommendations selecting

Leaving the number of guideline recommendations open-ended does not mean that any worthwhile sounding idea has been written up as a guideline. Although the number of different coding problems that could be encountered is infinite, an endless list of guidelines would be of little practical use. Worthwhile recommendations are those that minimize both the likelihood of faults being introduced by a developer or the effort needed by subsequent developers to comprehend the source code. Guideline recommendations covering situations that rarely occur in practice are wasted effort (not for the developers who rarely get to see them, but for the guideline author and tool vendors implementing checks for them).

These coding guidelines are not intended to recommend against the use of constructs that are obviously faults (i.e., developers have done something by mistake and would want to modify the code if the usage was pointed out to them). For instance, a guideline recommending against the use of uninitialized objects is equivalent to a guideline recommending against faults (i.e., pointless). Developers do not need to be given recommendations not to use these constructs. Guidelines either recommend against the use of constructs that are intentionally used (i.e., a developer did not use them by mistake) in a conforming program (any constructs

guidelines not faults

diagnostic 146 shall produce

that would cause a conforming translator to issue a diagnostic are not included), or they recommend that a particular implementation technique be used.

These guidelines deal with the use of C language constructs, not the design decisions behind their selection. It is not the intent to discuss how developers choose to solve the higher-level design and algorithmic issues associated with software development. These guidelines deal with instances of particular constructs at the source code level.

Source code faults are nearly always clichés; that is, developers tend to repeat the mistakes of others and their own previous mistakes. Not every instance of a specific construct recommended against by a guideline (e.g., an assignment operator in a conditional expression, `if (x = y)`) need result in a fault. However, because a sufficient number of instances have caused faults to occur in the past, it is considered to be worthwhile recommending against all usage of a construct.

Guidelines covering a particular construct cannot be considered in isolation from the rest of the language. The question has to be asked, of each guideline: "if developers are not allowed do this, what are they going to do instead?" A guideline that effectively forces developers into using an even more dangerous construct is a lot more than simply a waste of time. For instance, your authors experience is that placing too many restrictions on how enumerated constants are defined leads to developers using macro names instead— a counterproductive outcome.

Selecting guideline recommendations based on the preceding criteria requires both a detailed inventory of software faults for the C language (no distinction is made between faults that are detected in the source and faults that are detected as incorrect output from a program) and some measure of developer comprehension effort. Developer comprehension is discussed elsewhere. There have been relatively few reliable studies of software faults (Knuth's[753] log of faults in TEX is one such; see Fredericks[451] for a survey). Some of those that have been published have looked at faults that occur during initial development,[1344] and faults that occur during the evolution of an application, its maintenance period.[549, 1080]

developer 0 program comprehension

Guidelines that look worthy but lack empirical evidence for their cost effectiveness should be regarded with suspicion. The field of software engineering has a poor track record for experimental research. Studies[874, 1505] have found that most published papers in software related disciplines do not include any experimental validation. Whenever possible this book quotes results based on empirical studies (for the measurements by the author, either the raw data or the source code of the programs that generated the data are available from the author[1516]). Sometimes results from theoretical derivations are used. As a last resort, common practices and experience are sometimes quoted. Those studies that have investigated issues relating to coding practices have often used very inexperienced subjects (students studying at a university). The results of these inexperienced subject-based studies have been ignored.

experimental 0 studies

Table 0.2: Fault categories ordered by frequency of occurrence. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.[1080]

Rank	Fault Description	% Total Faults	Fix Rank	Rank	Fault Description	% Total Faults	Fix Rank
1	internal functionality	25.0	13	12	error handling	3.3	6
2	interface complexity	11.4	10	13	primitive's misuse	2.4	11
3	unexpected dependencies	8.0	4	14	dynamic data use	2.1	15
4	low-level logic	7.9	17	15	resource allocation	1.5	2
5	design/code complexity	7.7	3	16	static data design	1.0	19
6	other	5.8	12	17	performance	0.9	1
7	change coordinates	4.9	14	18	unknown interactions	0.7	5
8	concurrent work	4.4	9	19	primitives unsupported	0.6	19
9	race conditions	4.3	7	20	IPC rule violated	0.4	16
10	external functionality	3.6	8	21	change management complexity	0.3	21
11	language pitfalls i.e., use of = when == intended	3.5	18	22	dynamic data design	0.3	21

- A study by Thayer, Lipow, and Nelson[1344] looked at several Jovial (a Fortran-like language) projects during their testing phase. It was advanced for its time, using tools to analyze the source and being rigorous in the methodology of its detailed measurements. The study broke new ground: "Based on error histories seen in the data, define sets of error categories, both causative and symptomatic, to be applied in the analysis of software problem reports and their closure." Unfortunately, the quality of this work was not followed up by others and the level of detail provided is not sufficient for our needs here.
- Hatton[549] provides an extensive list of faults in C source code found by a static analysis tool. The tool used was an earlier version of one of the tools used to gather the usage information for this book. 0 Usage 1
- Perry[1080] looked at the modification requests for a 1 MLOC system that contained approximately 15% to 20% new code for each release. As well as counting the number of occurrences of each fault category, a weight was given to the effort required to fix them.

Table 0.3: Underlying cause of faults. The *none given* category occurs because sometimes both the fault and the underlying cause are the same. For instance, *language pitfalls*, or *low-level logic*. Based on Perry.[1080]

Rank	Cause Description	% Total Causes	Fix Rank
1	Incomplete/omitted design	25.2	3
2	None given	20.5	10
3	Lack of knowledge	17.8	8
4	Ambiguous design	9.8	9
5	Earlier incorrect fix	7.3	7
6	Submitted under duress	6.8	6
7	Incomplete/omitted requirements	5.4	2
8	Other	4.1	4
9	Ambiguous requirements	2.0	1
10	Incorrect modifications	1.1	5

Looking at the results (shown in Table 0.2) we see that although performance is ranked 17th in terms of number of occurrences, it moves up to first when effort to fix is taken into account. Resource allocation also moves up the rankings. The application measured has to operate in realtime, so performance and resource usage will be very important. The extent to which the rankings used in this case apply to other application domains is likely to depend on the application domain. Perry also measured the underlying causes (see Table 0.3) and the means of fault prevention (see Table 0.4).

Table 0.4: Means of fault prevention. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.[1080]

Rank	Means Description	% Observed	Fix Rank
1	Application walk-through	24.5	8
2	Provide expert/clearer documentation	15.7	3
3	Guideline enforcement	13.3	10
4	Requirements/design templates	10.0	5
5	Better test planning	9.9	9
6	Formal requirements	8.8	2
7	Formal interface specifications	7.2	4
8	Other	6.9	6
9	Training	2.2	1
10	Keep document/code in sync	1.5	7

- A study by Glass[496] looked at what he called *persistent software errors*. Glass appears to make an implicit assumption that faults appearing late in development or during operational use are somehow

different from those found during development. The data came from analyzing *software problem reports* from two large projects. There was no analysis of faults found in these projects during development.

Your author knows of no study comparing differences in faults found during early development, different phases of testing and operational use. Until proven otherwise, these Coding guideline subsections treat the faults found during different phases of development as having the same characteristics.

Usage 0 1

More detailed information on the usage of particular C constructs is given in the Usage sections of this book. While this information provides an estimate of the frequency-of-occurrence of these constructs, it does not provide any information on their correlation to occurrences of faults. These frequency of occurrence measurements were used in the decision process for deciding when particular constructs might warrant a guideline (the extent to which frequency of occurrence might affect developer performance. Note that power law of learning is not considered here.

power law 0 of learning

The selection of these guidelines was also influenced by the intended audience of developers, the types of programs they work on, and the priorities of the environment in which these developers work as follows:

- Developers are assumed to have imperfect memory, work in a fashion that minimizes their cognitive load, are not experts in C language and are liable to have incorrect knowledge about what they think C constructs mean; and have an incomplete knowledge base of the sources they are working on. Although there may be developers who are experts in C language and the source code they are working on, it is assumed here that such people are sufficiently rare that they are not statistically significant; in general these Coding guideline subsections ignore them. A more detailed discussion is given elsewhere.

coding 0 guidelines developers

- Applications are assumed to be large (over 50 KLOC) and actively worked on by more than one developer.

coding 0 guidelines applications

- Getting the software right is only one of the priorities in any commercial development group. Costs and time scales need to be considered. Following coding guidelines is sometimes a small component of what can also be a small component in a big project.

coding 0 guidelines cost drivers

9.2.1 Guideline recommendations must be enforceable

guideline recommendation enforceable

A guideline recommendation that cannot be enforced is unlikely to be of any use. Enforcement introduces several practical issues that constrain the recommendations made by guidelines, including the following:

- *Detecting violations.* It needs to be possible to deduce (by analyzing source code) whether a guideline is, or is not, being adhered to. The answer should always be the same no matter who is asking the question (i.e., the guidelines should be unambiguous).
- *Removing violations.* There needs to be a way of rewriting the source so that no guideline is violated. Creating a situation where it is not possible to write a program without violating one or other guidelines debases the importance of adhering to guidelines and creates a work environment that encourages the use of deviations.

guideline recommendation adherence has a reasonable cost

- *Testing modified programs.* Testing can be a very expensive process. The method chosen, by developers, to implement changes to the source may be based on minimizing the possible impact on other parts of a program, the idea being to reduce the amount of testing that needs to be done (or at least that appears to be needed to be done). Adhering to a guideline should involve an amount of effort that is proportional to the effort used to make changes to the source. Guidelines that could require a major source restructuring effort, after a small change to the source, are unlikely to be adhered to.

The procedures that might be followed in checking conformance to guidelines are not discussed in this book. A number of standards have been published dealing with this issue.[637, 638, 653]

A project that uses more than a handful of guidelines will find it uneconomical and impractical to enforce them without some form of automated assistance. Manually checking the source code against all guidelines is likely to be expensive and error prone (it could take a developer a working week simply to learn the guidelines, assuming 100 rules and 20 minutes study of each rule). Recognizing that some form of automated tool will be used, the wording for guidelines needs to be algorithmic in style.

There are situations where adhering to a guideline can get in the way of doing what needs to be done. Adhering to coding guidelines rarely has the highest priority in a commercial environment. Experience has shown that these situations can lead either to complete guideline recommendations being ignored, or be the thin end of the wedge that eventually leads to the abandonment of adherence to any coding guideline. The solution is to accept that guidelines do need to be broken at times. This fact should not be swept under the carpet, but codified into a deviation mechanism.

9.2.1.1 Uses of adherence to guidelines

While reducing the cost of ownership may be the aim of these guideline recommendations, others may see them as having other uses. For instance, from time to time there are calls for formal certification of source code to some coding guideline document or other. Such certification has an obvious commercial benefit to the certification body and any associated tools vendors. Whether such certification provides a worthwhile benefit to purchasers of software is debatable.[1439]

Goodhart's law[0.1] deals with the impact of external, human pressure on measurement and is applicable here. One of its forms is: "When a measure becomes a target, it ceases to be a good measure." Strathern[1307] describes how the use of a rating system changed the nature of university research and teaching.

Whether there is a greater economic benefit, to a company, in simply doing what is necessary to gain some kind of external recognition of conformance to a coding guideline document (i.e., giving little weight to the internal cost/benefit analysis at the source code level), or in considering adherence to guideline recommendations as a purely internal cost/benefit issue is outside the scope of this book.

9.2.1.2 Deviations

A list of possible deviations should be an integral part of any coding guideline. This list is a continuation of the experience and calculation that forms part of every guideline.

deviations
coding guidelines

The arguments made by the advocates of Total Quality Management[851] appear to be hard to argue against. The relentless pursuit of quality is to be commended for some applications, such as airborne systems and medical instruments. Even in other, less life-threatening, applications, quality is often promoted as a significant factor in enhancing customer satisfaction. Who doesn't want fault-free software? However, in these quality discussions, the most important factor is often overlooked— financial and competitive performance— (getting a product to market early, even if it contains known faults, is often much more important than getting a fault-free product to market later). Delivering a fault-free product to market late can result in financial ruin, just as delivering a fault prone product early to market. These coding guidelines aim of reducing the cost of software ownership needs to be weighed against the broader aim of creating value in a timely fashion. For instance, the cost of following a particular guideline may be much greater than normal, or an alternative technique may not be available. In these situations a strong case can be made for not adhering to an applicable guideline.

There is another practical reason for listing deviations. Experience shows that once a particular guideline has not been adhered to in one situation, developers find it easier not to adhere to it in other situations. Management rarely has access to anybody with sufficient expertise to frame a modified guideline (deviation) appropriate to the situation, even if that route is contemplated. Experience shows that developers rarely create a subset of an individual guideline to ignore; the entire guideline tends to be ignored. A deviation can stop adherence to a particular guideline being an all-or-nothing decision, helping to prevent the leakage of

[0.1] Professor Charles Goodhart, FBA, was chief adviser to the Bank of England and his "law" was originally aimed at financial measures (i.e., "As soon as the government attempts to regulate any particular set of financial assets, these become unreliable as indicators of economic trends.").

nonadherence. Deviations can provide an incremental sequence (increasing in cost and benefit) of decision points.

Who should decide when a deviation can be used? Both the authors of the source code and their immediate managers may have a potential conflict of interest with the longer-term goals of those paying for the development as follows:

- They may be under pressure to deliver a release and see use of a deviation as a short-cut.
- They may not be the direct beneficiaries of the investment being made in adhering to coding guidelines. Redirecting their resources to other areas of the project may seem attractive.
- They may not have the skill or resources needed to follow a guideline in a particular case. Admitting one's own limitations is always hard to do.

The processes that customers (which may be other departments within the same company) put in place to ensure that project managers and developers follow agreed-on practices are outside the scope of this book. Methods for processing deviation requests include:

developer 0 expertise

- referring all requests to an expert. This raises the question of how qualified a *C expert* must be to make technical decisions on deviations.
- making deviation decisions during code review.
- allowing the Q/A department to have the final say about which deviations are acceptable.

However, permission for the use of a deviation is obtained, all uses need to be documented. That is, each source construct that does not adhere to the full guideline, but a deviation of that guideline, needs to be documented. This documentation may simply be a reference to one location where the rationale for that deviation is given. Creating this documentation offers several benefits:

- It ensures that a minimum amount of thought has been given to the reasons for use of a deviation.
- It may provide useful information to subsequent developers. For instance, it can provide an indication of the number of issues that may need to be looked at when porting to a new translator, and the rationale given with a deviation can provide background information on coding decisions.
- It provides feedback to management on the practical implications of the guidelines in force. For instance, is more developer training required and/or should particular guidelines be reviewed (and perhaps reworded)?

Information given in the documentation for a deviation may need to include the following:

- The cost/benefit of following the deviation rather than the full guideline, including cost estimates.
- The risks associated with using the deviation rather than the full guideline recommendation.
- The alternative source code constructs and guidelines considered before selecting the deviation.

9.2.2 Code reviews

code reviews

Some coding guidelines are not readily amenable to automatic enforcement. This can occur either because they involve trade-offs among choices, or because commercial tool technology is not yet sufficiently advanced. The solution adopted here is to structure those guidelines that are not amenable to automatic enforcement so that they can be integrated into a code review process.

It is expected that those guideline recommendation capable of being automatically checked will have been enforced before the code is reviewed. Looking at the output of static analysis tools during code review is

usually an inefficient use of human resources. It makes sense for the developers writing the source code to use the analysis tools regularly, not just prior to reviews.

These coding guidelines are not intended to cover all the issues that should be covered during reviews. Problems with the specification, choice of algorithms, trade-offs in using constructs, agreement with the specification, are among the other issues that should be considered.

The impact of code reviews goes beyond the immediate consequences of having developers read and comment on each other's code. Knowing that their code is to be reviewed by others can affect developer's decision— making strategy. Even hypothetical questions raised during a code review can change subsequent decision making.[428]

0 justifying decisions

Code reviews are subject to the same commercial influences as other development activities; they require an investment of resources (a cost) to deliver benefits. Code reviews are widely seen as a good idea and are performed by many development groups. A very common rationale given for having code reviews is that they are a cost effective means of detecting faults. A recent review[1105] questioned this assumption, based on the lack of experimental evidence showing it to be true. Another reason for performing code reviews is the opportunity it provides for junior developers learn the culture of a development group.

Organizations that have a formal review procedure often follow a three-stage process of preparation, collection, and repair. During preparation, members of the review team read the source looking for as many defects as possible. During review the team as a whole looks for additional defects and collates a list of agreed-on defects. Repair is the resolution of these defects by the author of the source.

Studies by Porter, Siy, Mockuss, and Votta[1106–1108] to determine the best form for code reviews found that: inspection interval and effectiveness of defect detection were not significantly affected by team size (large vs. small), inspection interval and effectiveness of defect detection were not significantly affected by the number of sessions (single vs. multiple), and the effectiveness of defect detection was not improved by performing repairs between sessions of two-session inspections (however, inspection interval was significantly increased). They concluded that single-session inspections by small teams were the most efficient because their defect-detection rate was as good as other formats, and inspection interval was the same or less.

9.3 Relationship among guidelines

Individual guideline recommendations do not exist in isolation. They are collected together to form a set of coding guidelines. Several properties are important in a set of guideline recommendations, including:

coding guidelines relationship among

- It must be possible to implement the algorithmic functionality required by one guideline without violating any of the guidelines in a set.
- Consistency among guidelines within a set is a worthwhile aim.
- Being able to use the same process to enforce all requirements within a set of guidelines is a worthwhile aim.

As a complete set, the guideline recommendations in this book do not meet all of these requirements, but it is possible to create a number of sets that do meet them. It is management's responsibility to select the subset of guidelines applicable to their development situation.

9.4 How do guideline recommendations work?

How can adhering to these coding guidelines help reduce the cost of software ownership? The following are possible mechanisms:

guideline recommendations how they work

- Reduce the number of faults introduced into source code by recommending against the use of constructs known to have often been the cause of faults in the past. For instance, by recommending against the use of an assignment operator in a conditional expression, `if (x = y)`.
- Developers have different skills and backgrounds. Adhering to guidelines does not make developers write good code, but these recommendations can help prevent them from writing code that will be more costly than necessary to maintain.

- Developers' programming experience is often limited, so they do not always appreciate all the implications of using constructs. Guideline recommendations provide a prebuilt knowledge net. For instance, they highlight constructs whose behavior is not as immutable as developers might have assumed. The most common response your author hears from developers is "Oh, I didn't know that".

The primary purpose of coding guidelines is not usually about helping the original author of the code (although as a user of that code they can be of benefit to that person). Significantly more time and effort are spent maintaining existing programs than in writing new ones. For code maintenance, being able to easily extract information from source code, in order to predict the behavior of a program (sometimes called *program comprehension*), is an important issue.

Does reducing the cognitive effort needed to comprehend source code increase the rate at which developers comprehend it and/or reduce the number of faults they introduce into it? While there is no direct evidence proving that it does, these coding guideline subsections assume that it does.

9.5 Developer differences

developer differences

To what extent do individual developer differences affect the selection and wording of coding guidelines? To answer this question some of the things we would need to know include the following:

- the attributes that vary between developers,
- the number of developers (ideally the statistical distribution) having these different attributes and to what extent they possess them, and
- the affect these attribute differences have on developers' performance when working with source code.

Psychologists have been studying and measuring various human attributes for many years. These studies are slowly leading to a general understanding of how human cognitive processes operate. Unfortunately, there is no experimentally verified theory about the cognitive processes involved in software development. So while a lot of information on the extent of the variation in human attributes may be known, how these differences affect developers' performance when working with source code is unknown.

The overview of various cognitive psychology studies, appearing later in this introduction, is not primarily intended to deal with differences between developers. It is intended to provide a general description of the characteristics of the mental processing capabilities of the human mind. Strengths, weaknesses, and biases in these capabilities need to be addressed by guidelines. Sometimes the extent of individuals' capabilities do vary significantly in some areas. Should guidelines address the lowest common denominator (anybody could be hired), or should they assume a minimum level of capability (job applicants need to be tested to ensure they are above this level)?

What are the costs involved in recommending that the capabilities required to comprehend source code not exceed some maximum value? Do these costs exceed the likely benefits? At the moment these questions are somewhat hypothetical. There are no reliable means of measuring developers' different capabilities, as they relate to software development, and the impact of these capabilities on the economics of software development is very poorly understood. Although the guideline recommendations do take account of the capability limitations of developers, they are frustratingly nonspecific in setting boundaries.

developer 0 expertise

These guidelines assume some minimum level of knowledge and programming competence on the part of developers. They do not require any degree of expertise (the issue of expertise is discussed elsewhere).

- A study by Monaghan[956, 957] looked at measures for discriminating *ability* and *style* that are relevant to representational and strategy differences in people's problem solving.
- A study by Oberlander, Cox, Monaghan, Stenning, and Tobin[1023] investigated student responses to multimodal (more than one method of expression, graphical and sentences here) logic teaching. They found that students' preexisting cognitive styles affected both the teaching outcome and the structure of their logical discourse.

- A study by MacLeod, Hunt and Mathews[883] looked at sentence–picture comprehension. They found one group of subjects used a comprehension strategy that fit a linguistic model, while another group used a strategy that fit a pictorial–spatial model. A psychometric test of subjects showed a high correlation between the model a subject used and their spatial ability (but not their verbal ability). Sentence–picture comprehension is discussed in more detail elsewhere. In most cases C source visually appears, to readers, in a single mode, linear text. Although some tools are capable of displaying alternative representations of the source, they are not in widespread use. The extent to which a developer's primary mode of thinking may affect source code comprehension in this form is unknown.

934 sentence-picture relationships

The effect of different developer personalities is discussed elsewhere, as are working memory, reading span, rate of information processing, the affects of age, and cultural differences. Although most developers are male,[320] gender differences are not discussed.

0 developer personality
0 memory developer
1707 reading span
0 developer computational power
792 identifier information extraction
0 memory ageing
0 reasoning ability age-related
0 categorization cultural differences

9.6 What do these guidelines apply to?

A program (at least those addressed by these Coding guidelines) is likely to be built from many source files. Each source file is passed through eight phases of translation. Do all guidelines apply to every source file during every phase of translation? No, they do not. Guideline recommendations are created for a variety of different reasons and the rationale for the recommendation may only be applicable in certain cases; for instance:

coding guidelines what applied to?
108 source files
115 translation phases of
1866 preprocessing

- Reduce the cognitive effort needed to comprehend a program usually apply to the visible source code. That is, the source code as viewed by a reader, for example, in an editor. The result of preprocessing may be a more complicated expression, or sequence of nested constructs than specified by a guideline recommendation. But, because developers are not expected to have to read the output of the preprocessor, any complexity here may not be relevant,
- Common developer mistakes may apply during any phase of translation. The contexts should be apparent from the wording of the guideline and the construct addressed.
- Possible changes in implementation behavior can apply during any phase of translation. The contexts should be apparent from the wording of the guideline and the construct addressed.
- During preprocessing, the sequence of tokens output by the preprocessor can be significantly different from the sequence of tokens (effectively the visible source) input into it. Some guideline recommendations apply to the visible source, some apply to the sequence of tokens processed during syntax and semantic analysis, and some apply during other phases of translation.
- Different source files may be the responsibility of different development groups. As such, they may be subject to different commercial requirements, which can affect management's choice of guidelines applied to them.
- The contents of system headers are considered to be opaque and outside the jurisdiction of these guideline recommendations. They are provided as part of the implementation and the standard gives implementations the freedom to put more or less what they like into them (they could even contain some form of precompiled tokens, not source code). Developers are not expected to modify system headers.

121 header precompiled

- Macros defined by an implementation (e.g., specified by the standard). The sequence of tokens these macros expand to is considered to be opaque and outside the jurisdiction of these coding guidelines. These macros could be defined in system headers (discussed previously) or internally within the translator. They are provided by the implementation and could expand to all manner of implementation-defined extensions, unspecified, or undefined behaviors. Because they are provided by an implementation, the intended actual behavior is known, and the implementation supports it. Developers can use these macros at the level of functionality specified by the standard and not concern themselves with implementation details.

Applying these reasons in the analysis of source code is something that both automated guideline enforcement tools and code reviewers need to concern themselves with.

It is possible that different sets of guideline recommendations will need to be applied to different source files. The reasons for this include the following:

- The cost effectiveness of particular recommendations may change during the code's lifetime. During initial development, the potential savings may be large. Nearer the end of the application's useful life, the savings achieved from implementing some recommendations may no longer be cost effective.
- The cost effectiveness of particular coding guidelines may vary between source files. Source containing functions used by many different programs (e.g., application library functions) may need to have a higher degree of portability, or source interfacing to hardware may need to make use of representation information.
- The source may have been written before the introduction of these coding guidelines. It may not be cost effective to modify the existing source to adhere to all the guidelines that apply to newly written code.

It is management's responsibility to make decisions regarding the cost effectiveness of applying the different guidelines under differing circumstances.

Some applications contain automatically generated source code. Should these coding guidelines apply to this kind of source code? The answer depends on how the generated source is subsequently used. If it is treated as an invisible implementation detail (i.e., the fact that C is generated is irrelevant), then C guideline recommendations do not apply (any more than assembler guidelines apply to C translators that chose to generate assembler as an intermediate step on the way to object code). If the generated source is to be worked on by developers, just like human-written code, then the same guidelines should be applied to it as to human written code.

9.7 When to enforce the guidelines

coding guidelines when to enforce

Enforcing guideline recommendations as soon as possible (i.e., while developers are writing the code) has several advantages, including:

- Providing rapid feedback has been shown[582] to play an essential role in effective learning. Having developers check their own source provides a mechanism for them to obtain this kind of rapid feedback.
- Once code-related decisions have been made, the cost of changing them increases as time goes by and other developers start to make use of them.
- Developers' acceptance is increased if their mistakes are not made public (i.e., they perform the checking on their own code as it is written).

It is developers' responsibility to decide whether to check any modified source before using the compiler, or only after a large number of modifications, or at some other decision point. Checking in source to a version-control system is the point at which its adherence to guidelines stops being a private affair.

To be cost effective, the process of checking source code adherence to guideline recommendations needs to be automated. However, the state of the art in static analysis tools has yet to reach the level of sophistication of an experienced developer. Code reviews are the suggested mechanism for checking adherence to some recommendations. An attempt has been made to separate out those recommendations that are probably best checked during code review. This is not to say that these guideline recommendations should not be automated, only that your author does not think it is practical with current, and near future, static analysis technology.

The extent to which guidelines are automatically enforceable, using a tool, depends on the sophistication of the analysis performed; for instance, in the following (use of uninitialized objects is not listed as a guideline recommendation, but it makes for a simple example):

```
1  extern int glob;
2  extern int g(void);
3
4  void f(void)
5  {
6  int loc;
7
8  if (glob == 3)
9     loc = 4;
10 if (glob == 3)
11    loc++;         /* Does loc have a defined value here? */
12 if (glob == 4)
13    loc--;         /* Does loc have a defined value here? */
14 if (g() == 2)
15    loc = 9;
16 if (g() == glob)
17    ++loc;
18 }
```

The existing value of loc is modified when certain conditions are true. Knowing that it has a defined value requires analysis of the conditions under which the operations are performed. A static analysis tool might: (1) mark objects having been assigned to and have no knowledge of the conditions involved; (2) mark objects as assigned to when particular conditions hold, based on information available within the function that contains their definition; (3) the same as (2) but based on information available from the complete program.

9.8 Other coding guidelines documents

coding guidelines other documents

The writing of coding guideline documents is a remarkably common activity. Publicly available documents discussing C include,[440, 549, 595, 661, 724, 757, 912, 941, 942, 1096, 1097, 1131, 1135, 1142, 1274, 1305] and there are significantly more documents internally available within companies. Such guideline documents are seen as being a *good thing* to have. Unfortunately, few organizations invest the effort needed to write technically meaningful or cost-effective guidelines, they then fail to make any investment in enforcing them.[0.2]

The following are some of the creators of coding guideline include:

- *Software development companies.*[1234] Your author's experience with guideline documents written by development companies is that at best they contain well-meaning platitudes and at worse consist of a hodge-podge of narrow observations based on their authors' experiences with another language.
- *Organizations, user groups and consortia that are users of software.*[1109, 1479] Here the aim is usually to reduce costs for the organization, not software development companies. Coding guidelines are rarely covered in any significant detail and the material usually forms a chapter of a much larger document. Herrmann[568] provides a good review of the approaches to software safety and reliability promoted by the transportation, aerospace, defense, nuclear power, and biomedical industries through their published guidelines.
- *National and international standards.*[655] Perceived authority is an important attribute of any guidelines document. Several user groups and consortia are actively involved in trying to have their documents adopted by national, if not international, standards bodies. The effort and very broad spectrum of consensus needed for publication as an International Standard means that documents are likely to be first adopted as National Standards.

The authors of some coding guideline documents see them as a way of making developers write good programs (whatever they are). Your author takes the view that adherence to guidelines can only help prevent mistakes being made and reduce subsequent costs.

[0.2] If your author is told about the existence of coding guidelines while visiting a company's site, he always asks to see a copy; the difficulty his hosts usually have in tracking down a copy is testament to the degree to which they are followed.

Most guideline recommendations specify subsets, not supersets, of the language they apply to. The term *safe subset* is sometimes used. Perhaps this approach is motivated by the idea that a language already has all the constructs it needs, the desire not to invent another language, or simply an unwillingness to invest in the tools that would be needed to handle additional constructs (e.g., adding strong typing to a weakly typed language). The guidelines in this book have been written as part of a commentary on the C Standard. As such, they restrict themselves to constructs in that document and do not discuss recommendations that involve extensions.

Experience with more strongly typed languages suggests that strong typing does detect some kinds of faults before program execution. Although experimental tool support for stronger type checking of C source is starting to appear,[866, 1004, 1219] little experience in its use is available for study. This book does not specify any guideline recommendations that require stronger type checking than that supported by the C Standard.

Quite a few coding guideline documents have been written for C++.[279, 561, 798, 928–930, 943, 1051, 1098, 1318] It is interesting to note that these coding guideline documents concentrate almost exclusively on the object-oriented features of C++ (i.e., primarily those constructs not available in C). It is almost as if their authors believe that developers using C++ will not make any of the mistakes that C developers make, despite one language almost being a superset of the other.

Coding guideline documents for other languages include those for Ada,[270, 655] Cobol,[1013] Fortran,[772] PERL,[271] Prolog,[293] and SQL.[420]

9.8.1 Those that stand out from the crowd

The aims and methods used to produce coding guidelines documents vary. Many early guideline documents concentrated on giving advice to developers about how to write efficient code.[772] The availability of powerful processors, coupled with large quantities of source code, has changed the modern (since the 1980s) emphasis to one of maintainability rather than efficiency. When efficiency is an issue, the differences between processors and compilers makes it difficult to give general recommendations. Vendors' reference manuals sometimes provide useful background advice.[26, 628] The Object Defect Classification[228] covers a wide variety of cases and has been shown to give repeatable results when used by different people.[388]

9.8.1.1 Bell Laboratories and the 5ESS

measurements 5ESS

Bell Laboratories undertook a root-cause analysis of faults in the software for their 5ESS Switching System.[1497] The following were found to be the top three causes of faults, and their top two subcomponents:

1. Execution/oversight— 38%, which in turn was broken down into inadequate attention to details (75%) and inadequate consideration to all relevant issues (11%).
2. Resource/planning— 19%, which in turn was broken down into not enough engineer time (76%) and not enough internal support (4%).
3. Education/training— 15%, which in turn was broken down into area of technical responsibility (68%) and programming language usage (15%).

In an attempt to reduce the number of faults, a set of "Code Fault Prevention Guidelines" and a "Coding Fault Inspection Checklist" were written and hundreds of engineers were trained in their use. These guideline recommendations were derived from more than 600 faults found in a particular product. As such, they could be said to be tuned to that product (nothing was said about how different root causes might evolve over time).

Based on measurements of previous releases of the 5ESS software and engineering cost per house to implement the guidelines (plus other bug inject countermeasures), it was estimated that for an investment of US$100 K, a saving of US$7 M was made in product rework and testing.

One of the interesting aspects of programs is that they can contain errors in logic and yet continue to perform their designated function; that is, faults in the source do not always show up as a perceived fault by the user of a program. Static analysis of code provides an estimate of the number of potential faults, but not all of these will result in reported faults.

Why did the number of faults reported in the 5ESS software drop after the introduction of these guideline recommendations? Was it because previous root causes were a good measure of future root-cause faults?

The guideline recommendations created do not involve complex constructs that required a deep knowledge of C. They are essentially a list of mistakes made by developers who had incomplete knowledge of C. The recommendations could be looked on as C language knowledge tuned to the reduction of faults in a particular application program. The coding guideline authors took the approach that it is better to avoid a problem area than expect developers to have detailed knowledge of the C language (and know how to deal with problem areas).

In several places in the guideline document, it is pointed out that particular faults had costly consequences. Although evidence that adherence to a particular set of coding guidelines would have prevented a costly fault provides effective motivation for the use of those recommendations, this form of motivation (often seen in coding guideline documents) is counter-productive when applied to individual guideline recommendations. There is rarely any evidence to show that the reason for a particular coding error being more expensive that another one is anything other than random chance.

9.8.1.2 MISRA

MISRA (Motor Industry Software Reliability Association, http://www.misra.org.uk) published a set of *Guidelines for the use of the C language in vehicle based software*.[941, 942] These guideline recommendations were produced by a committee of interested volunteers and have become popular in several domains outside the automobile industry. For the most part, they are based on the implementation-defined, undefined, and unspecified constructs listed in Annex G of the C90 Standard. The guidelines relating to issues outside this annex are not as well thought through (the technicalities of what is intended and the impact of following a guideline recommendation). MISRA

There are now 15 or more vendors who offer products that claim to enforce compliance to the MISRA guidelines. At the time of this writing these tools are not always consistent in their interpretation of the wording of the guidelines. Being based on volunteer effort, MISRA does not have the resources to produce a test suite or provide timely responses to questions concerning the interpretation of particular guidelines.

9.8.2 Ada

Although the original purpose of the Ada language was to reduce total software ownership costs, its rigorous type checking and handling of runtime errors subsequently made it, for many, the language of choice for development of high-integrity systems. An ISO Technical Report[655] (a TR does not have the status of a standard) was produced to address this market. 0 Ada using

The rationale given in many of the *Guidance* clauses of this TR is that of making it possible to perform static analysis by recommending against the use of constructs that make such analysis difficult or impossible to perform. Human factors are not explicitly mentioned, although this could be said to be the major issue in some of the constructs discussed. Various methods are described as not being cost effective. The TR gives the impression that what it proposes is cost effective, although no such claim is made explicitly.

> *. . . , it can be seen that there are four different reasons for needing or rejecting particular language features within this context:* ISO/IEC TR 15942:2000
>
> 1. *Language rules to achieve predictability,*
> 2. *Language rules to allow modelling,*
> 3. *Language rules to facilitate testing,*
> 4. *Pragmatic considerations.*

This TR also deals with the broader issues of verification techniques, code reviews, different forms of static analysis, testing, and compiler validation. It recognizes that developers have different experience levels and sometimes (e.g., clause 5.10.3) recommends that some constructs only be used by experienced developers (nothing is said about how experience might be measured).

9.9 Software inspections

software inspections introduction

Software inspections, technical reviews, program walk-throughs (whatever the name used), all involve people looking at source code with a view to improving it. Some of the guidelines in this book are specified for enforcement during code reviews, primarily because automated tools have not yet achieved the sophistication needed to handle the constructs described.

Software inspections are often touted as a cost-effective method of reducing the number of defects in programs. However, their cost effectiveness, compared to other methods, is starting to be questioned. For a survey of current methods and measurements, see;[793] for a detailed handbook on the subject, see.[452]

Reading inspection
Reading 770
eye movement

During inspections a significant amount of time is spent reading — reading requirements, design documents, and source code. The cost of, and likely mistakes made during, code reading are factors addressed by some guideline recommendations. The following are different ways of reading source code, as it might be applied during code reviews:

- *Ad hoc reading techniques.* This is a catch-all term for those cases, very common in commercial environments, where the software is simply given to developers. No support tools or guidance is given on how they should carry out the inspection, or what they should look for. This lack of support means that the results are dependent on the skill, knowledge, and experience of the people at the meeting.
- *Checklist reading.* As its name implies this reading technique compares source code constructs against a list of issues. These issues could be collated from faults that have occurred in the past, or published coding guidelines such as the ones appearing in this book. Readers are required to interpret applicability of items on the checklist against each source code construct. This approach has the advantage of giving the reader pointers on what to look for. One disadvantage is that it constrains the reader to look for certain kinds of problems only.
- *Scenario-based reading.* Like checklist reading, scenario-based reading provides custom guidance.[939] However, as well as providing a list of questions, a scenario also provides a description on how to perform the review. Each scenario deals with the detection of the particular defects defined in the custom guidance. The effectiveness of scenario-based reading techniques depends on the quality of the scenarios.
- *Perspective-based reading.* This form of reading checks source code from the point of view of the customers, or consumers, of a document.[98] The rationale for this approach is that an application has many different stakeholders, each with their own requirements. For instance, while everybody can agree that software quality is important, reaching agreement on what the attributes of quality are can be difficult (e.g., timely delivery, cost effective, correct, maintainable, testable). Scenarios are written, for each perspective, listing activities and questions to ask. Experimental results on the effectiveness of perspective-based reading of C source in a commercial environment are given by Laitenberger and Jean-Marc DeBaud.[792]
- *Defect-based reading.* Here different people focus on different defect classes. A scenario, consisting of a set of questions to ask, is created for each defect class; for instance, invalid pointer dereferences might be a class. Questions to ask could include; Has the lifetime of the object pointed to terminated? Could a pointer have the null pointer value in this expression? Will the result of a pointer cast be correctly aligned?
- *Function-point reading.* One study[794] that compared checklist and perspective-based reading of code, using professional developers in an industrial context, found that perspective-based reading had a lower cost per defect found.

This book does not recommend any particular reading technique. It is hoped that the guideline recommendations given here can be integrated into whatever method is chosen by an organization.

10 Applications

coding guidelines applications

Several application issues can affect the kind of guideline recommendations that are considered to be applicable. These include the application domain, the economics behind the usage, and how applications evolve over time. These issues are discussed next.

The use of C as an intermediate language has led to support for constructs that simplify the job of translation from other languages. Some of these constructs are specified in the standard (e.g., a trailing comma in initializer lists), while others are provided as extensions (e.g., `gcc`'s support for taking the address of labels and being able to specify the **`register`** storage class on objects declared with file scope, has influenced the decision made by some translator implementors, of other languages to generate C rather than machine code[345]).

1641 initialization syntax

10.1 Impact of application domain

Does the application domain influence the characteristics of the source code? This question is important because frequency of occurrence of constructs in source is one criterion used in selecting guidelines. There are certainly noticeable differences in language usage between some domains; for instance:

0 Usage 1

- *Floating point.* Many applications make no use of any floating-point types, while some scientific and engineering applications make heavy use of this data type.
- *Large initializers.* Many applications do not initialize objects with long lists of values, while the device driver sources for the Linux kernel contain many long initializer lists.

There have been studies that looked at differences within different industries (e.g., banking, aerospace, chemical[551]). It is not clear to what extent the applications measured were unique to those industries (e.g., some form of accounting applications will be common to all of them), or how representative the applications measured might be to specific industries as a whole.

Given the problems associated with obtaining source code for the myriad of different application domains, and the likely problems with separating out the effects of the domain from other influences, your author decided to ignore this whole issue. A consequence of this decision is that these guideline recommendations are a union of the possible issues that can occur across all application domains. Detailed knowledge of the differences would be needed to build a set of guidelines that would be applicable to each application domain. Managers working within a particular application domain may want to select guidelines applicable to that domain.

10.2 Application economics

Coding guidelines are applicable to applications of all sizes. However, there are economic issues associated with the visible cost of enforcing guideline recommendations. For instance, the cost of enforcement is not likely to be visible when writing new code (the incremental cost is hidden in the cost of writing the code). However, the visible cost of ensuring that a large body of existing, previously unchecked, code can be significant.

The cost/benefit of adhering to a particular guideline recommendation will be affected by the economic circumstances within which the developed application sits. These circumstances include

- short/long expected lifetime of the application,
- relative cost of updating customers,
- quantity of source code,
- acceptable probability of application failure (adherence may not affect this probability, but often plays well in any ensuing court case), and
- expected number of future changes/updates.

There are so many possible combinations that reliable estimates of the effects of these issues, on the applicability of particular guidelines, can only be made by those involved in managing the development projects (the COCOMO cost-estimation model uses 17 cost factors, 5 scale factors, a domain-specific factor, and a count of the lines of code in estimating the cost of developing an application). The only direct economic issues associated with guidelines, in this book, we discussed earlier and through the choice of applications measured.

COCOMO 0
development 0
context
Usage 0
1

10.3 Software architecture

software architecture

The term *architecture* is used in a variety of software development contexts.[0.3] The analogy with buildings is often made, "firm foundations laying the base for . . . ". This building analogy suggests a sense of direction and stability. Some applications do have these characteristics (in particular many of those studied in early software engineering papers, which has led to the view that most applications are like this). Many large government and institutional applications have this form (these applications are also the source of the largest percentage of published application development research).

To remind readers, the primary aim of these coding guidelines is to minimize the cost of software ownership. Does having a good architecture help achieve this aim? Is it possible to frame coding guidelines that can help in the creation of good architecture? What is a good architecture?

What constitutes good software architecture is still being hotly debated. Perhaps it is not possible to predict in advance what the best architecture for a given application is. However, experience shows that in practice the customer can rarely specify exactly what it is they want in advance, and applications close to what they require are obviously not close enough (or they would not be paying for a different one to be written). Creating a good architecture, for a given application, requires knowledge of the whole and designers who know how to put together the parts to make the whole. In practice applications are very likely to change frequently; it might be claimed that applications only stop changing when they stop being used. Experience has shown that it is almost impossible to predict the future direction of application changes.

The conclusion to be drawn, for these observations, is that there are reasons other than incompetence for applications not to have any coherent architecture (although at the level of individual source files and functions this need not apply). In a commercial environment, profitability is a much stronger motive than the desire for coherent software architecture.

Software architecture, in the sense of organizing components into recognizable structures, is relevant to reading and writing source in that developers' minds also organize the information they hold. People do not store information in long-term memory as unconnected facts. These coding guidelines assume that having programs structured in a way that is compatible with how information is organized in developers' minds, and having the associations between components of a program correspond to how developers make associations between items of information, will reduce the cognitive effort of reading source code. The only architectural and organizational issues considered important by the guideline recommendations in this book are those motivated by the characteristics of developers' long-term memory storage and retrieval.

memory 0
developer
categorization 0

For a discussion of the pragmatics of software architecture, see Foote.[434]

10.3.1 Software evolution

application evolution

Applications that continue to be used tend to be modified over time. The term *software evolution* is sometimes used to describe this process. Coding guidelines are intended to reduce the costs associated with modifying source. What lessons can be learned from existing applications that have evolved?

There have been several studies that looked at the change histories of some very large (several million

[0.3] Some developers like to refer to themselves as software architects. In the UK such usage is against the law, " . . . punishable by a fine not exceeding level 4 on the standard scale . . . " (Architects Act 1997, Part IV):

Use of title "architect".

20. – (1) A person shall not practise or carry on business under any name, style or title containing the word "architect" unless he is a person registered under this Act.

(2) Subsection (1) does not prevent any use of the designation "naval architect", "landscape architect" or "golf-course architect".

line,[469] or a hundred million[382]) programs over many years,[382, 532, 1031] and significant growth over a few years.[499] Some studies have simply looked at the types of changes and their frequency. Others have tried to correlate faults with the changes made. None have investigated the effect of source characteristics on the effort needed to make the changes.

The one thing that is obvious from the data published to date: Researchers are still in the early stages of working out which factors are associated with software evolution.

- A study[330] at Bell Labs showed the efficiency gains that could be achieved using developers who had experience with previous releases over developers new to a project. The results indicated that developers who had worked on previous releases spent 20% of their time in project discovery work. This 20% was put down as the cost of working on software that was evolving (the costs were much higher for developers not familiar with the project). 0 software development expertise
- Another Bell Labs study[952] looked at predicting the risk of introducing a fault into an existing software system while performing an update on it. They found that the main predictors were the number of source lines affected, developer experience, time needed to make the change, and an attribute they called *diffusion*. Diffusion was calculated from the number of subsystems, modules, and files modified during the change, plus the number of developers involved in the work. Graves[516] also tried to predict faults in an evolving application. He found that the fault potential of a module correlated with a weighted sum of the contributions from all the times the module had been changed (recent changes having the most weight). Similar findings were obtained by Ohlsson.[1030, 1031]
- Lehman has written a number of papers[832] on what he calls the *laws of software evolution*. Although they sound plausible, these "laws" are based on empirical findings from relatively few projects.
- Kemerer and Slaughter[721] briefly review existing empirical studies and also describe the analysis of 25,000 change events in 23 commercial software systems (Cobol-based) over a 20-year period.
- Other studies have looked at the interaction of module coupling and cohesion with product evolution. 1821 coupling and cohesion

11 Developers

The remainder of this coding guidelines subsection has two parts. This first major subsection discusses the tasks that developers perform, the second (the following major subsection) is a review of psychology studies carried out in human characteristics of relevance to reading and writing source code. There is an academic research field that goes under the general title *the psychology of programming*; few of the research results from this field have been used in this book for reasons explained elsewhere. However, without being able to make use of existing research applicable to commercial software development, your author has been forced into taking this two-part approach; which is far from ideal. A consequence of this approach is that it is not possible to point at direct experimental evidence for some of the recommendations made in coding guidelines. The most that can be claimed is that there is a possible causal link between specific research results, cognitive theories, and some software development activities.

coding guidelines developers

0 psychology of programming

Although these coding guidelines are aimed at a particular domain of software development, there is no orientation toward developers having any particular kinds of mental attributes. It is hoped that this discussion will act as a stimulus for research aimed at the needs of commercial software development, which cannot take place unless commercial software developers are willing to give up some of their time to act as subjects (in studies). It is hoped that this book will persuade readers of the importance of volunteering to take part in this research.

0 Usage 1
0 developer differences

11.1 What do developers do?

In this book, we are only interested in developer activities that involve source code. Most studies,[1078] the time spent on these activities does not usually rise above 25%, of the total amount of time developers spend on all activities. The non-source code-related activities, the other 75%, are outside the scope of this

developers what do they do?

book. In this book, the reason for reading source code is taken to be that developers want to comprehend program behavior sufficiently well to be able to make changes to it. Reading programs to learn about software development, or for pleasure, are not of interest here.

The source that is eventually modified may be a small subset of the source that has been read. Developers often spend a significant amount of their time working out what needs to be modified and the impact the changes will have on existing code.[330]

The tools used by developers to help them search and comprehend source tend to be relatively unsophisticated.[1238] This general lack of tool usage needs to be taken into account in that some of the tasks performed in a *manual*-comprehension process will be different from those carried out in a tool-assisted process.

cognitive effort 0
cognitive load 0

The following properties are taken to be important attributes of source code, because they affect developer cognitive effort and load:

reading 770
kinds of

- *Readable.* Source is both scanned, looking for some construct, and read in a booklike fashion. The symbols appearing in the visible source need to be arranged so that they can be easily seen, recognized, and processed.
- *Comprehensible.* Having read a sequence of symbols in the source, their meaning needs to be comprehended.
- *Memorable.* With applications that may consist of many thousands of line of source code (100 KLOC is common), having developers continually rereading what they have previously read because they have forgotten the information they learned is not cost effective. Cognitive psychology has yet to come up with a model of human memory that can be used to calculate the memorability of source code. One practical approach might be to measure developer performance in reconstructing the source of a translation unit (an idea initially proposed by Shneiderman,[1230] who proposed a 90–10 rule—a competent developer should be able to reconstruct functionally 90% of a translation unit after 10 minutes of study).

developer 0
training

- *Unsurprising.* Developers have expectations. Meeting those expectations reduces the need to remember special cases, and it reduces the possibility of faults caused by developers making assumptions (not checking that their expectations are true).

For a discussion of the issues involved in collecting data on developers' activities and some findings, see Dewayne[1079] and Bradac.[149]

11.1.1 Program understanding, not

developer program comprehension

One of the first tasks a developer has to do when given source code is figure out what it does (the word *understand* is often used by developers). What exactly does it mean to understanding a program? The word *understanding* can be interpreted in several different ways; it could imply

- knowing all there is to know about a program. Internally (the source code and data structures) and externally— its execution time behavior.
- knowing the external behavior of a program (or perhaps knowing the external behavior in a particular environment), but having a limited knowledge of the internal behavior.
- knowing the internal details, but having a limited knowledge of the external behavior.

The concept of *understanding a program* is often treated as being a yes/no affair. In practice, a developer will know more than nothing and less than everything about a program. Source code can be thought of as a web of knowledge. By reading the source, developers acquire beliefs about it; these beliefs are influenced by their existing beliefs. Existing beliefs (many might be considered to be knowledge rather than belief, by the person holding them) can involve a programming language (the one the source is written in), general computing algorithms, and the application domain.

belief maintenance 0

When reading a piece of source code for the first time, a developer does not start with an empty set of beliefs. Developers will have existing beliefs, which will affect the interpretation given to the source code read. Developers learn about a program, a continuous process without a well-defined ending. This learning process involves the creation of new beliefs and the modification of existing ones. Using a term (*understanding*) that implies a yes/no answer is not appropriate. Throughout this book, the term *comprehension* is used, not *understanding*.

Program comprehension is not an end in itself. The purpose of the investment in acquiring this knowledge (using the definition of knowledge as "belief plus complete conviction and conclusive justification") is for the developer to be in a position to be able predict the behavior of a program sufficiently well to be able to change it. Program comprehension is not so much knowledge of the source code as the ability to predict the effects of the constructs it contains (developers do have knowledge of the source code; for instance, knowing which source file contains a declaration).

While this book does not directly get involved in theories of how people learn, program comprehension is a learning process. There are two main theories that attempt to explain learning. Empirical learning techniques look for similarities and differences between positive and negative examples of a concept. Explanation-based learning techniques operate by generalizing from a single example, proving that the example is an instance of the concept. The proof is constructed by an inference process, making use of a domain theory, a set of facts, and logical implications. In explanation-based learning, generalizations retain only those attributes of an example that are necessary to prove the example is an instance of the concept. Explanation-based learning is a general term for learning methods, such as knowledge compilation and chunking, that create new concepts that deductively follow from existing concepts. It has been argued that a complete model of concept learning must have both an empirical and an explanation-based component.

What strategies do developers use when trying to build beliefs about (comprehend) a program? The theories that have been proposed can be broadly grouped into the following:

- *The top-down approach.* The developer gaining a top-level understanding of what the program does. Once this is understood, the developer moves down a level to try to understanding the components that implement the top level. This process is repeated for every component at each level until the lowest level is reached. A developer might chose to perform a depth-first or width-first analysis of components.
- *The bottom-up approach.* This starts with small sequences of statements that build a description of what they do. These descriptions are fitted together to form higher-level descriptions, and so on, until a complete description of the program has been built.
- *The opportunistic processors approach.* Here developers use both strategies, depending on which best suits the purpose of what they are trying to achieve.[840]

There have been a few empirical studies, using experienced (in the industrial sense) subjects, of how developers comprehend code (the purely theoretically based models are not discussed here). Including:

- A study by Letovsky[839] asked developers to talk aloud (their thoughts) as they went about the task of adding a new feature to a program. He views developers as *knowledge base understanders* and builds a much more thorough model than the one presented here.
- A study by Littman, Pinto, Letovsky and Soloway[861] found two strategies in use by the developers (minimum of five years experience) they observed: In a systematic strategy the developers seek to obtain information about how the program behaves before modifying it; and in an as-needed strategy developers tried to minimize the effort needed to study the program to be modified by attempting to localize those parts of a program where the changes needed to be made. Littman et al. found that those developers using the systematic strategy outperformed those using the as-needed strategy for the 250-line program used in the experiment. They also noted the problems associated with attempting to use the systematic strategy with much larger programs.

- A study by Pennington[1072] investigated the differences in comprehension strategies used by developers who achieved high and low levels of program comprehension. Those achieving high levels of comprehension tended to think about both the application domain and the program (source code) domain rather than just the program domain. Pennington[1073] also studied mental representations of programs; for small programs she found that professional programmers built models based on control flow rather than data flow.
- A study by von Mayrhauser and Vans[1429, 1430] looked at experienced developers maintaining large, 40,000+ LOC applications and proposed an integrated code comprehension model. This model contained four major components, (1) program model, (2) situated model, (3) top-down model, and (4) knowledge base.
- A study by Shaft and Vessey[1217] gave professional programmer subjects source code from two different application domains (accounting and hydrology). The subjects were familiar with one of the domains but not the other. Some of the subjects used a different comprehension strategy for the different domains.

11.1.1.1 Comprehension as relevance

relevance

Programming languages differ from human languages in that they are generally viewed, by developers, as a means of one-way communication with a computer. Human languages have evolved for interactive communication between two, or more, people who share common ground.[0.4]

One of the reasons why developers sometimes find source code comprehension so difficult is that the original authors did not write it in terms of a communication with another person. Consequently, many of the implicit assumptions present in human communication may not be present in source code. Relevance is a primary example. Sperber and Wilson[1271] list the following principles of human communication:

Sperber and Wilson[1271]

Principle of relevance

1. *Every act of ostensive communication communicates a presumption of its own optimal relevance.*

Presumption of optimal relevance

1. *The set of assumptions **I** which the communicator intends to make manifest to the addressee is relevant enough to make it worth the addressee's while to process the ostensive stimulus.*
2. *The ostensive stimulus is the most relevant one the communicator could have used to communicate **I**.*

program image 141

A computer simply executes the sequence of instructions contained in a program image. It has no conception of application assumptions and relevance. The developer knows this and realizes that including such information in the code is not necessary. A common mistake made by novice developers is to assume that the computer is aware of their intent and will perform the appropriate operations. Teaching developers to write code such that can be comprehended by two very different addressee's is outside the scope of these coding guidelines.

Source code contains lots of details that are relevant to the computer, but often of little relevance to a developer reading it. Patterns in source code can be used as indicators of relevance; recognizing these patterns is something that developers learn with experience. These coding guidelines do not discuss the teaching of such recognition.

Developers often talk of the *intended meaning* of source code, i.e., the meaning that the original author of the code intended to convey. Code comprehension being an exercise in obtaining an intended meaning that is assumed to exist. However, the only warranted assumption that can be made about source code is that the operations specified in it contribute to a meaning.

[0.4] The study of meaning and communication between people often starts with Grice's maxims,[520] but readers might find Sperber and Wilson[1271] easier going.

11.1.2 The act of writing software

The model of developers sitting down to design and then write software on paper, iterating through several versions before deciding their work is correct, then typing it into a computer is still talked about today. This method of working may have been necessary in the past because access to computer terminals was often limited and developers used paper implementations as a method of optimizing the resources available to them (time with, and without, access to a computer).

Much modern software writing is done sitting at a terminal, within an editor. Often no written, paper, notes are used. Everything exists either in the developer's head or on the screen in front of him (or her). However, it is not the intent of this book to suggest alternative working practices. Changing a system that panders to people's needs for short-term gratification,[450] to one that delays gratification and requires more intensive periods of a difficult, painful activity (thinking) is well beyond your author's capabilities.

Adhering to guideline recommendation does not guarantee that high quality software will be written; it can only help reduce the cost of ownership of the software that is written.

These coding guidelines assume that the cost of writing software is significantly less than the cost of developer activities that occur later (testing, rereading, and modification by other developers). Adhering to guideline may increase the cost of writing software. The purpose of this investment is to make savings (which are greater than the costs by an amount proportional to the risk of the investment) in the cost of these later activities. 0 ROI

It is hoped that developers will become sufficiently fluent in using these guideline recommendations and that they will be followed automatically while entering code. A skilled developer should aim to be able to automatically perform as much of the code-writing process as possible. Performing these tasks automatically frees up cognitive resources for use on other problems associated with code development.

It is a profoundly erroneous truism . . . that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead (1861–1947)

It is not suggested that the entire software development process take place without any thinking. The process of writing code can be compared to writing in longhand. The writer thinks of a sentence and his hand automatically writes the words. It is only schoolchildren who need to concentrate on the actual process of writing the words. 0 developer flow

11.2 Productivity

Although much talked about, there has been little research on individual developer productivity. There is the often quoted figure of a 25-to-1 productivity difference between developers; however, this is a misinterpretation of figures presented in two tables of a particular paper.[514] Hopefully the analysis by Prechelt[1117] will finally put a stop to researchers quoting this large, incorrect, figure. The differences in performance found by Prechelt are rarely larger than four, similar to the performance ranges found by the original research. productivity developer

Few measurement programs based on individual developers have been undertaken; many measures are based on complete projects, dividing some quantity (often lines of code) by the number of individuals working on them. See Scacchi[1197] for a review of the empirical software productivity research and Jones[680] provides a good discussion of productivity over the complete life cycle of a project. However, some of the issues discussed (e.g., response time when editing source) are rooted in a mainframe environment and are no longer relevant.

Are there any guideline recommendations that the more productive developers use that we can all learn from? Your author knows of no published research that investigates productivity at this level of detail. Age-related productivity issues[881, 1252] are not discussed in these coding guidelines. The subject of expertise is discussed elsewhere. 0 expertise

12 The new(ish) science of people

It is likely that the formal education of this book's readership will predominantly have been based on the so-called *hard sciences*. The word *hard* being used in the sense of having theories backed by solid experimental results, which are repeatable and have been repeated many times. These sciences, and many engineering disciplines, have also been studied experimentally for a long period of time. The controversies surrounding the basic theory, taught to undergraduates, have been worked through.

Psychology has none of those advantages. There are often unseen, complex interactions going on inside the object being studied (people's responses to questions and problems). Because of this, studies using slightly different experimental situations can obtain very different results. The field is also relatively new, and the basic theory is still being argued over. Consequently, this book cannot provide a definitive account of the underlying theories relating to the subject of immediate interest here— reading and writing source code.

The results of studies, and theories, from psychology are starting to become more widely applied in other fields. For instance, economists are starting to realize that people do not always make rational decisions.[1229] Researchers are also looking at the psychology of programming.

cognitive psychology

The subfield of psychology that is of most relevance to this book is cognitive psychology. The goal of cognitive psychology is to understand the nature of human intelligence and how it works. Other subfields include clinical psychology (understanding why certain thought malfunctions occur) and social psychology (how people behave in groups or with other individuals).[0.5]

12.1 Brief history of cognitive psychology

Topics of interest to cognitive psychology were discussed by the Greeks as part of their philosophical thinking. This connection with philosophy continued through the works of Descartes, Kant, Mill, and others. In 1879, Wilhelm Wundt established the first psychology laboratory in Germany; this date is considered to mark the start of psychology as an independent field. Wundt believed that the workings of the mind were open to self-observation. The method involved introspection by trained observers under controlled conditions. Unfortunately, different researchers obtained different results from these introspection experiments, so the theory lost creditability.

During the 1920s, John Watson and others developed the theory known as *Behaviorism*. This theory was based on the idea that psychology should be based on external behavior, not on any internal workings of the mind. The theory is best known through its use of rats in various studies. Although widely accepted in the US for a long time, behaviorism was not so dominant in Europe, where other theories were also developed.

Measurements on human performance were given a large boost by World War II. The introduction of technology, such as radar, required people to operate it. Information about how people were best trained to use complex equipment, and how they could best maintain their attention on the job at hand, was needed.

Cognitive psychology grew into its current form through work carried out between 1950 and 1970. The inner workings of the mind were center stage again. The invention of the computer created a device, the operation of which was seen as a potential parallel for the human mind. Information theory as a way of processing information started to be used by psychologists. Another influence was linguistics, in particular Noam Chomsky's theories for analyzing the structure of language. The information-processing approach to cognitive psychology is based on carrying out experiments that measured human performance and building models that explained the results. It does not concern itself with actual processes within the brain, or parts of the brain, that might perform these functions.

Since the 1970s, researchers have been trying to create theories that explain human cognition in terms of how the brain operates. These theories are known as *cognitive architectures*. The availability of brain scanners (which enable the flow of blood through the brain to be monitored, equating blood flow to activity) in the 1990s has created the research area of cognitive neuroscience, which looks at brain structure and processes.

[0.5] For a good introduction to the subject covering many of the issues discussed here, see either *Cognitive Psychology: A Student's Handbook* by Eysenck and Keane[405] or *Cognitive Psychology and its Implications* by Anderson.[33]

12.2 Evolutionary psychology

evolutionary psychology

Human cognitive processes are part of the survival package that constitutes a human being. The cognitive processes we have today exist because they increased (or at least did not decrease) the likelihood of our ancestors passing on their genes thorough offspring. Exactly what edge these cognitive processes gave our ancestors, over those who did not possess them, is a new and growing area of research known as *evolutionary psychology*. To quote one of the founders of the field:[286]

Cosmides[286]

> *Evolutionary psychology is an approach to psychology, in which knowledge and principles from evolutionary biology are put to use in research on the structure of the human mind. It is not an area of study, like vision, reasoning, or social behavior. It is a way of thinking about psychology that can be applied to any topic within it.*
>
> *. . . all normal human minds reliably develop a standard collection of reasoning and regulatory circuits that are functionally specialized and, frequently, domain-specific. These circuits organize the way we interpret our experiences, inject certain recurrent concepts and motivations into our mental life, and provide universal frames of meaning that allow us to understand the actions and intentions of others. Beneath the level of surface variability, all humans share certain views and assumptions about the nature of the world and human action by virtue of these human universal reasoning circuits.*

These functionally specialized circuits (the theory often goes by the name of the *massive modularity hypothesis*) work together well enough to give the impression of a powerful, general purpose processor at work. Because they are specialized to perform a given task when presented with a problem that does not have the expected form (the use of probabilities rather than frequency counts in the conjunction fallacy) performance is degraded (peoples behavior appears incompetent, or even irrational, if presented with a reasoning problem). The following are the basic principles:

0 conjunction fallacy

Cosmides[286]

> *Principle 1. The brain is a physical system. It functions as a computer. Its circuits are designed to generate behavior that is appropriate to your environmental circumstances.*
>
> *Principle 2. Our neural circuits were designed by natural selection to solve problems that our ancestors faced during our species' evolutionary history.*
>
> *Principle 3. Consciousness is just the tip of the iceberg; most of what goes on in your mind is hidden from you. As a result, your conscious experience can mislead you into thinking that our circuitry is simpler than it really is. Most problems that you experience as easy to solve are very difficult to solve— they require very complicated neural circuitry.*
>
> *Principle 4. Different neural circuits are specialized for solving different adaptive problems.*
>
> *Principle 5. Our modern skulls house a stone age mind.*

Although this field is very new and has yet to establish a substantial body of experimental results and theory, it is referred to throughout these coding guidelines. The standard reference is Barkow, Cosmides, and Tooby[94] (Mithen[945] provides a less-technical introduction).

12.3 Experimental studies

experimental studies

Much of the research carried out in cognitive psychology has used people between the ages of 18 and 21, studying some form of psychology degree, as their subjects. There has been discussion by psychology researchers on the extent to which these results can be extended to the general populace.[92] However, here we are interested in the extent to which the results obtained using such subjects is applicable to how developers behave?

Given that people find learning to program difficult, and there is such a high failure rate for programming courses[768] it is likely that some kind of ability factors are involved. However, because of the lack of studies investigating this issue, it is not yet possible to know what these programming ability factors might be. There are a large number of developers who did not study for some form of a computing degree at university, so the fact that experimental subjects are often students taking other kinds of courses is unlikely to be an issue.

12.3.1 The importance of experiments

The theories put forward by the established sciences are based on experimental results. Being elegant is not sufficient for a theory to be accepted; it has to be backed by experiments.

Software engineering abounds with theories, and elegance is often cited as an important attribute. However, experimental results for these theories are often very thin on the ground. The computing field is evolving so rapidly that researchers do not seem willing to invest significant amounts of their time gathering experimental data when there is a high probability that many of the base factors will have completely changed by the time the results are published.

Replication is another important aspect of scientific research; others should be able to duplicate the results obtained in the original experiment. Replication of experiments within software research is relatively rare; possible reasons include

- the pace of developments in computing means that there are often more incentives for trying new ideas rather than repeating experiments to verify the ideas of others,
- the cost of performing an experiment can be sufficiently high that the benefit of replication is seen as marginal, and/or
- the nature of experiments involving large-scale, commercial projects are very difficult to replicate. Source code can be duplicated perfectly, so there is no need to rewrite the same software again.

A good practical example of the benefits of replication and the dangers of not doing any is given by Brooks.[162] Another important issue is the statistical power of experiments.[938] Experiments that fail can be as important as those that succeed. Nearly all published, computing-related papers describe successes. The benefits of publishing negative results (i.e., ideas that did not work) has been proposed by Prechelt.[1116] A study[1249] of 5,453 papers published in software engineering journals between 1993 and 2002 found that only 1.9% reported controlled experiments (of which 72.6% used students only as subjects) and even then the statistical power of these experiments fell below expected norms.[373]

12.4 The psychology of programming

psychology of programming

Studies on the psychology of programming have taken their lead from trends in both psychology and software engineering. In the 1960s and 1970s, studies attempted to measure performance times for various tasks. Since then researchers have tried to build models of how people carry out the tasks involved with various aspects of programming.

Several theories about how developers go about the task of comprehending source code have been proposed. There have also been specific proposals about how to reduce developer error rates, or to improve developer performance. Unfortunately, the experimental evidence for these theories and proposals is either based on the use of inexperienced subjects or does not include sufficient data to enable statistically significant conclusions to be drawn. A more detailed, critical analysis of the psychological study of programming is given by Sheil[1222] (the situation does not seem to have changed since this paper was written 20 years ago).

Several studies have investigated how novices write software. This is both an area of research interest and of practical use in a teaching environment. The subjects taking part in these studies also have the characteristics of the population under investigation (i.e., predominantly students). However, this book is aimed at developers who have several years experience writing code; it is not aimed at novices and it does not teach programming skills.

Lethbridge, Sim, and Singer[838] discuss some of the techniques used to perform field studies inside software companies.

12.4.1 Student subjects

Although cognitive psychology studies use university students as their subjects there is an important characteristic they generally have, for these studies, that they don't have for software development studies.[1248] That characteristic is experience— that is, years of practice performing the kinds of actions (e.g., reading text,

making decisions, creating categories, reacting to inputs) they are asked to carry out in the studies. However, students, typically, have very little experience of writing software, perhaps 50 to 150 hours. Commercial software developers are likely to have between 1,000 to 10,000 hours of experience. A study by Moher and Schneider[953] compared the performance of students and professional developers in program comprehension tasks. The results showed that experience was a significant predictor of performance level (greater than aptitude in this study).

Reading and writing software is a learned skill. Any experiments that involve a skill-based performance need to take into account the subjects' skill level. The coding guidelines in this book are aimed at developers in a commercial environment where it is expected that they will have at least two years experience in software development.

Use of very inexperienced developers as subjects in studies means that there is often a strong learning effect in the results. Student subjects taking part in an experiment often get better at the task because they are learning as they perform it. Experienced developers have already acquired the skill in the task being measured, so there is unlikely to be any significant learning during the experiment. An interesting insight into the differences between experiments involving students and professional developers is provided by a study performed by Basili[98] and a replication of it by Ciolkowski.[237]

A note on differences in terminology needs to be made here. Many studies in the psychology of programming use the phrase *expert* to apply to a subject who is a third-year undergraduate or a graduate student (the term *novice* being applied to first-year undergraduates). In a commercial software development environment a recent graduate is considered to be a *novice* developer. Somebody with five or more years of commercial development experience might know enough to be called an *expert*.

12.4.2 Other experimental issues

When an experiment is performed, it is necessary to control all variables except the one being measured. It is also necessary to be able to perform the experiments in a reasonable amount of time. Most commercial programs contain thousands of lines of source code. Nontrivial programs of this size can contain any number of constructs that could affect the results of an experiment; they would also require a significant amount of effort to read and comprehend. Many experiments use programs containing less than 100 lines of source. In many cases, it is difficult to see how results obtained using small programs will apply to much larger programs.

The power of the statistical methods used to analyze experimental data depends on the number of different measurements made. If there are few measurements, the statistical significance of any claim's results will be small. Because of time constraints many experiments use a small number of different programs, sometimes a single program. All that can be said for any results obtained for a single program is that the results apply to that program; there is no evidence of generalization to any other programs.

Is the computer language used in experiments significant? The extent to which the natural language, spoken by a person, affects their thinking has been debated since Boas, Sapir, and Whorf developed the linguistic relativity hypothesis[871]. In this book, we are interested in C, a member of the procedural computer language family. More than 99.9% of the software ever written belongs to languages in this family. However, almost as many experiments seem to use nonprocedural languages, as procedural ones. Whether the language family of the experiment affects the applicability of the results to other language families is unknown. However, it will have an effect on the degree of believability given to these results by developers working in a commercial environment.

792 language affecting thought

12.5 What question is being answered?

Many of the studies carried out by psychologists implicitly include a human language (often English) as part of the experiment. Unless the experiments are carefully constructed, unexpected side-effects may be encountered. These can occur because of the ambiguous nature of words in human language, or because of subjects expectations based on their experience of the nature of human communication.

The following three subsections describe famous studies, which are often quoted in introductory cognitive psychology textbooks. Over time, these experiments have been repeated in various, different ways and the

underlying assumptions made by the original researchers has been challenged. The lesson to be learned from these studies is that it can be very difficult to interpret a subject's answer to what appears to be a simple question. Subjects simply may not have the intellectual machinery designed to answer the question in the fashion it is phrased (base rate neglect), they may be answering a completely different question (conjunction fallacy), or they may be using a completely unexpected method to solve a problem (availability heuristic).

12.5.1 Base rate neglect

base rate neglect
representa-0
tive heuristic

Given specific evidence, possible solutions to a problem can be ordered by the degree to which they are representative of that evidence (i.e., their probability of occurring as the actual solution, based on past experience). While these representative solutions may appear to be more likely to be correct than less-representative solutions, for particular cases they may in fact be less likely to be the solution. Other factors, such as the prior probability of the solution, and the reliability of the evidence can affect the probability of any solution being correct.

A series of studies, Kahneman and Tversky[706] suggested that subjects often seriously undervalue the importance of prior probabilities (i.e., they neglected base-rates). The following is an example from one of these studies. Subjects were divided into two groups, with one group of subjects being presented with the following cover story:

> A panel of psychologists have interviewed and administered personality tests to 30 engineers and 70 lawyers, all successful in their respective fields. On the basis of this information, thumbnail descriptions of the 30 engineers and 70 lawyers have been written. You will find on your forms five descriptions, chosen at random from the 100 available descriptions. For each description, please indicate your probability that the person described is an engineer, on a scale from 0 to 100.

and the other group of subjects presented with identical cover story, except the prior probabilities were reversed (i.e., they were told that the personality tests had been administered to 70 engineers and 30 lawyers). Some of the descriptions provided were designed to be compatible with the subjects' stereotype of engineers, others were designed to be compatible with the stereotypes of lawyers, and one description was intended to be neutral. The following are two of the descriptions used.

> Jack is a 45-year-old man. He is married and has four children. He is generally conservative, careful and ambitious. He shows no interest in political and social issues and spends most of his free time on his many hobbies which include home carpentry, sailing, and mathematical puzzles.
>
> The probability that Jack is one of the 30 engineers in the sample of 100 is ____%.

> Dick is a 30-year-old man. He is married with no children. A man of high ability and high motivation, he promises to be quite successful in his field. He is well liked by his colleagues.
>
> The probability that Dick is one of the 70 lawyers in the sample of 100 is ____%.

Following the five descriptions was this null description.

> Suppose now that you are given no information whatsoever about an individual chosen at random from the sample.
>
> The probability that this man is one of the 30 engineers in the sample of 100 is ____%.

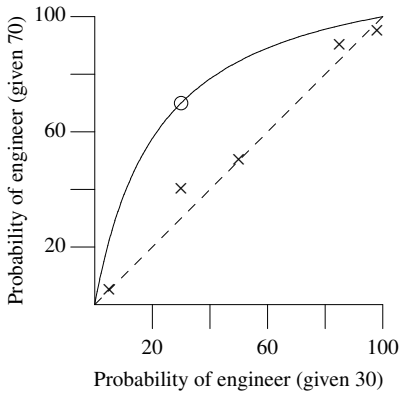


Figure 0.10: Median judged probability of subjects choosing an engineer, for five descriptions and for the null description (unfilled circle symbol). Adapted from Kahneman.[706]

In both groups, half of the subjects were asked to evaluate, for each description, if the person described was an engineer. The other subjects were asked the same question, except they were asked about lawyers.

The probability of a person being classified as an engineer, or lawyer, can be calculated using Bayes' theorem. Assume that, after reading the description, the estimated probability of that person being an engineer is P. The information that there are 30 engineers and 70 lawyers in the sample allows us to modify the estimate, P, to obtain a more accurate estimate (using all the information available to us). The updated probability is $0.3P/(0.3P + 0.7(1 - P))$. If we are told that there are 70 engineers and 30 lawyers, the updated probability is $0.7P/(0.7P + 0.3(1 - P))$. For different values of the estimate P, we can plot a graph using the two updated probabilities as the x and y coordinates. If information on the number of engineers and lawyers is not available, or ignored, the graph is a straight line.

The results (see Figure 0.10) were closer to the straight line than the Bayesian line. The conclusion drawn was that information on the actual number of engineers and lawyers in the sample (the base-rate) had minimal impact on the subjective probability chosen by subjects.

Later studies[756] found that peoples behavior when making decisions that included a base-rate component was complex. Use of base-rate information was found to depend on how problems and the given information was framed (large between study differences in subject performance were also seen). For instance, in some cases subjects were found to use their own experiences to judge the likelihood of certain events occurring rather than the probabilities given to them in the studies. In some cases the ecological validity of using Bayes' theorem to calculate the probabilities of outcomes has been questioned.

To summarize: while people have been found to ignore base-rates when making some decisions, this behavior is far from being universally applied to all decisions.

12.5.2 The conjunction fallacy

An experiment originally performed by Tversky and Kahneman[1375] presented subjects with the following problem. conjunction fallacy

> Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations.
>
> Please rank the following statements by their probability, using 1 for the most probable and 8 for the least probable.
>
> (a) Linda is a teacher in elementary school.
>
> (b) Linda works in a bookstore and takes Yoga classes.

(c) Linda is active in the feminist movement.
(d) Linda is a psychiatric social worker.
(e) Linda is a member of the League of Women Voters.
(f) Linda is a bank teller.
(g) Linda is an insurance sales person.
(h) Linda is a bank teller and is active in the feminist movement.

In a group of subjects with no background in probability or statistics, 89% judged that statement (h) was more probable than statement (f). Use of simple mathematical logic shows that Linda cannot be a feminist bank teller unless she is also a bank teller, implying that being only a bank teller is at least as likely, if not more so, than being both a bank teller and having some additional attribute. When the subjects were graduate students in the decision science program of the Stanford Business School (labeled as statistically sophisticated by the experimenters), 85% judged that statement (h) was more probable than statement (f).

These results (a compound event being judged more probable than one of its components) have been duplicated by other researchers performing different experiments. A recent series of studies[1233] went as far as checking subjects' understanding of the word *probability* and whether statement (f) might be interpreted to mean *Linda is a bank teller and not active in the feminist movement* (it was not).

This pattern of reasoning has become known as *the conjunction fallacy*.

On the surface many of the subjects in the experiment appear to be reasoning in a nonrational way. How can the probability of the event *A and B* be greater than the probability of event *A*? However, further studies have found that the likelihood of obtaining different answers can be affected by how the problem is expressed. The effects of phrasing the problem in terms of either *probability* or *frequency* were highlighted in a study by Fiedler.[421] The original Tversky and Kahneman study wording was changed to the following:

There are 100 people who fit the description above. How many of them are:
(a) bank tellers?
(b) bank tellers and active in the feminist movement?
. . .

In this case, only 22% of subjects rated the *bank teller and active in the feminist movement* option as being more frequent than the *bank teller* only option. When Fiedler repeated the experiment using wording identical to the original Tversky and Kahneman experiment, 91% of subjects gave the feminist bank teller option as more probable than the bank teller only option. A number of different explanations, for the dependence of the conjunction fallacy on the wording of the problem, have been proposed.

evolutionary psychology 0

Evolutionary psychologists have interpreted these results as showing that people are not very good at reasoning using probability. It is argued that, in our daily lives, events are measured in terms of their frequency of occurrence (e.g., how many times fish were available at a particular location in the river). This event-based measurement includes quantity, information not available when probabilities are used. Following this argument through suggests that the human brain has become specialized to work with frequency information, not probability information.

conjunction fallacy pragmatic interpretation relevance 0

Hertwig and Gigerenzer[569] point out that, in the Linda problem, subjects were not informed that they were taking part in an exercise in probability. Subjects therefore had to interpret the instructions; in particular, what did the experimenter mean by *probability*? Based on Grice's[520] theory of conversational reasoning, they suggested that the likely interpretation given to the word *probability* would be along the lines of "something which, judged by present evidence, is likely to be true, to exist, or to happen," (one of the Oxford English dictionary contemporary definitions of the word), not the mathematical definition of the word.

Grice's theory was used to make the following predictions:

Prediction 1: Probability judgments. If asked for probability judgments, people will infer its nonmathematical meanings, and the proportion of conjunction violations will be high as a result.

Prediction 2: Frequency judgments. If asked for frequency judgments, people will infer mathematical meanings, and the proportion of conjunction violations will decrease as a result.

Prediction 3: Believability judgments. If the term "probability" is replaced by "believability", then the proportion of conjunction violations should be about as prevalent as in the probability judgment.

A series of experiments confirmed these predictions. A small change in wording caused subjects to have a completely different interpretation of the question.

12.5.3 Availability heuristic

availability heuristic

How do people estimate the likelihood of an occurrence of an event? The availability heuristic argues that, in making an estimate, people bring to mind instances of the event; the more instances brought to mind, the more likely it is to occur. Tversky and Kahneman[1373] performed several studies in an attempt to verify that people use this heuristic to estimate probabilities. Two of the more well-known experiments follow.

The first is judgment of word frequency; here subjects are first told that.

> The frequency of appearance of letters in the English language was studied. A typical text was selected, and the relative frequency with which various letters of the alphabet appeared in the first and third positions in words was recorded. Words of less than three letters were excluded from the count.
>
> You will be given several letters of the alphabet, and you will be asked to judge whether these letters appear more often in the first or in the third position, and to estimate the ratio of the frequency with which they appear in these positions.

They were then asked the same question five times, using each of the letters (K, L, N, R, V).

> Consider the letter R.
> Is R more likely to appear in:
>
> - the first position?
> - the third position? (check one)
>
> My estimate for the ratio of these two values is ___:1.

Of the 152 subjects, 105 judged the first position to be more likely (47 the third position more likely). The median estimated ratio was 2:1.

In practice, words containing the letter *R* in the third position occur more frequently in texts than words with *R* in the first position. This is true for all the letters— *K*, *L*, *N*, *R*, *V*.

The explanation given for these results was that subjects could more easily recall words beginning with the letter *R*, for instance, than recall words having an *R* as the third letter. The answers given, being driven by the availability of instances that popped into the subjects' heads, not by subjects systematically counting all the words they knew.

An alternative explanation of how subjects might have reached their conclusion was proposed by Sedlmeier, Hertwig, and Gigerenzer.[1210] First they investigated possible ways in which the availability heuristic might operate; Was it based on availability-by-number (the number of instances that could be recalled) or availability-by-speed (the speed with which instances can be recalled). Subjects were told (the following is an English translation, the experiment took place in Germany and used German students) either:

> Your task is to recall as many words as you can in a certain time. At the top of the following page you will see a letter. Write down as many words as possible that have this letter as the first (second) letter.

or,

> Your task is to recall as quickly as possible one word that has a particular letter as the first (second) letter. You will hear first the position of the letter and then the letter. From the moment you hear the letter, try to recall a respective word and verbalize this word.

Subjects answers were used to calculate an estimate of relative word frequency based on either availability-by-number or on availability-by-speed. These relative frequencies did not correlate with actual frequency of occurrence of words in German. The conclusion drawn was that the availability heuristic was not an accurate estimator of word frequency, and that it could not be used to explain the results obtained by Tversky and Kahneman.

If subjects were not using either of these availability heuristics, what mechanism are they using? Jonides and Jones[691] have shown, based on a large body of results, that subjects are able to judge the number of many kinds of events in a way that reflects the actual relative frequencies of the events with some accuracy.

Sedlmeier et al.[1210] proposed (what they called the *regressed-frequencies hypothesis*) that (a) the frequencies with which individual letters occur at different positions in words are monitored (by people while reading), and (b) the letter frequencies represented in the mind are regressed toward the mean of all letter frequencies. This is a phenomenon often encountered in frequency judgment tasks, where low frequencies tend to be overestimated and high frequencies underestimated; although this bias affects the accuracy of the absolute size of frequency judgments, it does not affect their rank order. Thus, when asked for the relative frequency of a particular letter, subjects should be expected to give judgments of relative letter frequencies that reflect the actual ones, although they will overestimate relative frequencies below the mean and underestimate those above the mean — a simple regressed-frequency heuristic. The studies performed by Sedlmeier et al. consistently showed subjects' judgments conforming best to the predictions of the regressed-frequencies hypothesis.

While it is too soon to tell if the regressed-frequencies hypothesis is the actual mechanism used by subjects, it does offer a better fit to experimental results than the availability heuristic.

13 Categorization

categorization

Children as young as four have been found to use categorization to direct the inferences they make,[480] and many different studies have shown that people have an innate desire to create and use categories (they have also been found to be sensitive to the costs and benefits of using categories[884]). By dividing items in the world into categories of things, people reduce the amount of information they need to learn[1112] by building an indexed data structure that will enable them to lookup information on specific items they may not have encountered before (by assigning that item to one or more categories and extracting information common to items in those categories). For instance, a flying object with feathers and a beak might be assigned to the category *bird*, which suggests the information that it lays eggs and may be migratory.

Source code is replete with examples of categories; similar functions are grouped together in the same source file, objects belonging to a particular category are defined as members of the same structure type, and enumerated types are defined to represent a common set of symbolic names.

translation unit 1810 (syntax)
structure type 530 (sequentially allocated objects)
declaration 1348 (syntax)
enumeration 517 (set of named constants)
symbolic name 822

People seem to have an innate desire to create categories (people have been found to expect random sequences to have certain attributes,[407] e.g., frequent alternation between different values, which from a mathematical perspective represent regularity). There is the danger that developers, reading a program's source code will create categories that the original author was not aware existed. These *new* categories may

represent insights into the workings of a program, or they may be completely spurious (and a source of subsequent incorrect assumptions, leading to faults being introduced).

Categories can be used in many thought processes without requiring significant cognitive effort (a built-in operation). For instance, categorization can be used to perform inductive reasoning (the derivation of generalized knowledge from specific instances), and to act as a memory aid (remembering the members of a category). There is a limit on the cognitive effort that developers have available to be used and making use of a powerful ability, which does not require a lot of effort, helps optimize the use of available resources.

There have been a number of studies[1166] looking at how people use so-called *natural categories* (i.e., those occurring in nature such as mammals, horses, cats, and birds) to make inductive judgments. People's use of categorical-based arguments (e.g., "Grizzly bears love onions." and "Polar bears love onions." therefore "All bears love onions.") has also been studied.[1037]

Source code differs from nature in that it is created by people who have control over how it is organized. Recognizing that people have an innate ability to create and use categories, there is a benefit in trying to maximize positive use (developers being able to infer probable behaviors and source code usage based on knowing small amounts of information) of this ability and to minimize negative use (creating unintended categories, or making inapplicable inductive judgments).

Source code can be organized in a myriad of ways. The problem is finding the optimal organization, which first requires knowing what needs to be optimized. For instance, I might decide to split some functions I have written that manipulate matrices and strings into two separate source files. I could decide that the functions I wrote first will go in the first file and those that I wrote later in the second file, or perhaps the first file will contain those functions used on project X and the second file those functions used on project Y. To an outside observer, a more *natural* organization might be to place the matrix-manipulation functions in the first file and the string-manipulation functions in the second file.

In a project that grows over time, functions may be placed in source files on an as-written basis; a maintenance process that seeks to minimize disruption to existing code will keep this organization. When two separate projects are merged into one, a maintenance process that seeks to minimize disruption to existing code is unlikely to reorganize source file contents based on the data type being manipulated. This categorization process, based on past events, is a major factor in the difficulty developers have in comprehending *old* source. Because category membership is based on historical events, developers either need knowledge of those events or they have to memorize information on large quantities of source. Program comprehension changes from using category-based induction to relying on memory for events or source code.

Even when the developer is not constrained by existing practices the choice of source organization is not always clear-cut. An organization based on the data type being manipulated is one possibility, or there may only be a few functions and an organization based on functionality supported (i.e., printing) may be more appropriate. Selecting which to use can be a difficult decision. The following subsections discuss some of the category formation studies that have been carried out, some of the theories of category formation, and possible methods of calculating similarity to category.

Situations where source code categorization arise include: deciding which structure types should contain which members, which source files should contain which object and function definitions, which source files should be kept in which directories, whether functionality should go in a single function or be spread across several functions, and what is the sequence of identifiers in an enumerated type?

1810 declarations in which source file

Explicitly organizing source code constructs so that future readers can make use of their innate ability to use categories, to perform inductive reasoning, is not meant to imply that other forms of reasoning are not important. The results of deductive reasoning are generally the norm against which developer performance is measured. However, in practice, developers do create categories and use induction. Coding guidelines need to take account of this human characteristic. Rather than treating it as an aberration that developers need to be trained out of, these coding guidelines seek to adapt to this innate ability.

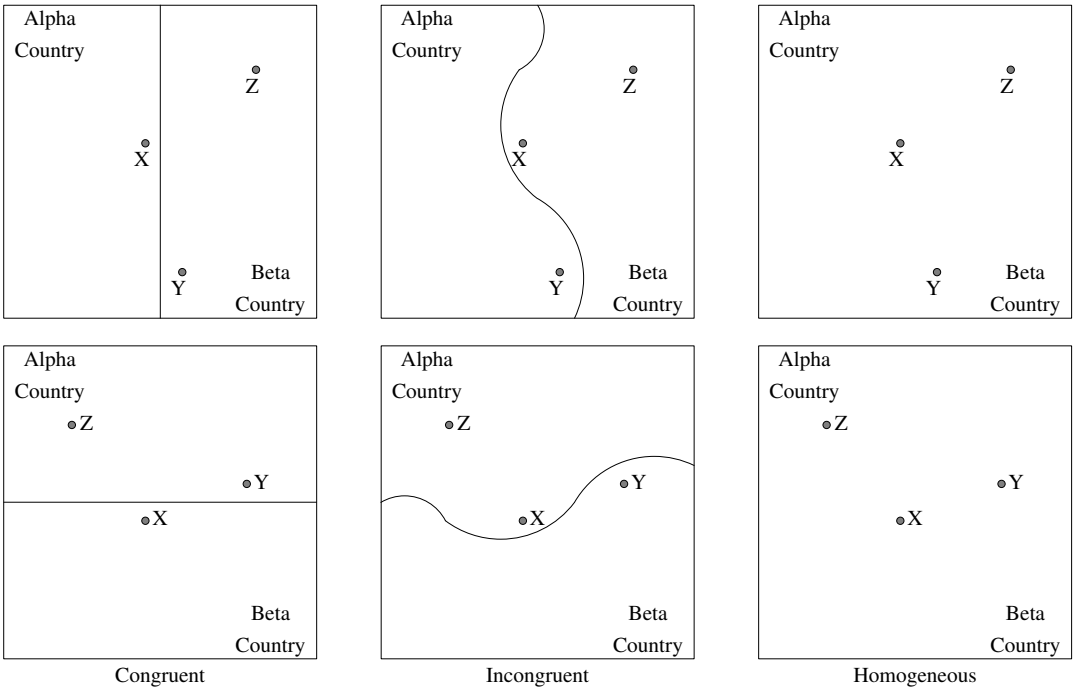


Figure 0.11: Country boundaries distort judgment of relative city locations. Adapted from Stevens.[1295]

13.1 Category formation

How categories should be defined and structured has been an ongoing debate within all sciences. For instance, the methods used to classify living organisms into family, genus, species, and subspecies has changed over the years (e.g., most recently acquiring a genetic basis).

Categories do not usually exist in isolation. Category judgment is often organized according to a hierarchy of relationships between concepts— a taxonomy. For instance, Jack Russell, German Shepherd, and Terrier belong to the category of dog, which in turn belongs to the category of mammal, which in turn belongs to the category of living creature. Organizing categories into hierarchies means that an attribute of a higher-level category can affect the perceived attributes of a subordinate category. This effect was illustrated in a study by Stevens and Coupe.[1295] Subjects were asked to remember the information contained in a series of maps (see Figure 0.11). They were then asked questions such as: "Is X east or west of Y?", and "Is X north or south of Y?" Subjects gave incorrect answers 18% of the time for the congruent maps, but 45% of the time for the incongruent maps (15% for the homogeneous). They were using information about the relative locations of the countries to answer questions about the city locations.

Several studies have shown that people use around three levels of abstraction in creating hierarchical relationships. Rosch[1179] called the highest level of abstraction the *superordinate-level*— for instance, the general category furniture. The next level down is the *basic-level*; this is the level at which most categorization is carried out— for instance, car, truck, chair, or table. The lowest level is the *subordinate-level*, denoting specific types of objects. For instance, a family car, a removal truck, my favourite armchair, a kitchen table. Rosch found that the basic-level categories had properties not shared by the other two categories; adults spontaneously name objects at this level. It is also the abstract level that children acquire first, and category members tend to have similar overall shapes.

- A study by Markman and Wisniewski[896] investigated how people view superordinate-level and basic-level categories as being different. The results showed that basic-level categories, derived from the

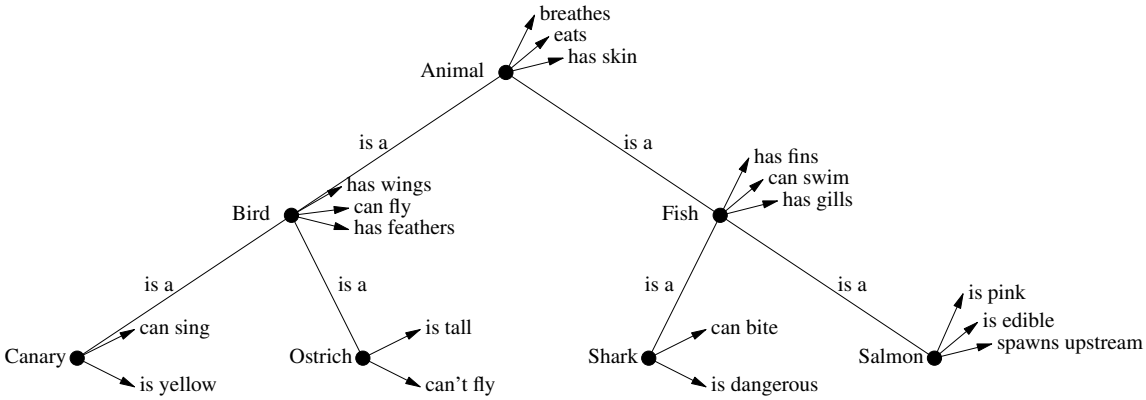


Figure 0.12: Hypothetical memory structure for a three-level hierarchy. Adapted from Collins.[258]

same superordinate-level, had a common structure that made it easy for people to compare attributes; for instance, motorcycle, car, and truck are basic-level categories of vehicle. They all share attributes (so-called *alignable differences*), for instance, number of wheels, method of steering, quantity of objects that can be carried, size of engine, and driver qualifications that differ but are easily compared. Superordinate-level categories differ from each other in that they do not share a common structure. This lack of a common structure means it is not possible to align their attributes to differentiate them. For these categories, differentiation occurs through the lack of a common structure. For instance, the superordinate-level categories — vehicle, musical instrument, vegetable, and clothing — do not share a common structure.

- A study by Tanaka and Taylor[1331] showed that the quantity of a person's knowledge and experience can affect the categories they create and use.
- A study by Johansen and Palmeri[671] showed that representations of perceptual categories can change with categorization experience. While these coding guidelines are aimed at experienced developers, they recognize that many experienced developers are likely to be inexperienced comprehenders of much of the source code they encounter. The guidelines in this book take the default position that, given a choice, they should assume an experienced developer who is inexperienced with the source being read.

There are likely to be different ways of categorizing the various components of source code. These cases are discussed in more detail elsewhere. Commonality and regularities shared between different sections of source code may lead developers to implicitly form categories that were not intended by the original authors. The extent to which the root cause is poor categorization by the original developers, or simply unrelated regularities, is not discussed in this book.

530 structure type
sequentially allocated objects
1629 typedef name
syntax
517 enumeration
set of named constants
1348 declaration
visual layout
1707 statement
visual layout

What method do people use to decide which, if any, category a particular item is a member of? Several different theories have been proposed and these are discussed in the following subsections.

13.1.1 The Defining-attribute theory

The defining-attribute theory proposes that members of a category are characterized by a set of defining attributes. This theory predicts that attributes should divide objects up into different concepts whose boundaries are well defined. All members of the concept are equally representative. Also, concepts that are a basic-level of a superordinate-level concept will have all the attributes of that superordinate level; for instance, a sparrow (small, brown) and its superordinate bird (two legs, feathered, lays eggs); see Figure 0.12.

Although scientists and engineers may create and use defining-attribute concept hierarchies, experimental evidence shows that people do not naturally do so. Studies have shown that people do not treat category

members as being equally representative, and some are rated as more typical than others.[1168] Evidence that people do not structure concepts into the neat hierarchies required by the defining-attribute theory was provided by studies in which subjects verified membership of a more distant superordinate more quickly than an immediate superordinate (according to the theory, the reverse situation should always be true).

13.1.2 The Prototype theory

In this theory, categories have a central description, the prototype, that represents the set of attributes of the category. This set of attributes need not be necessary, or sufficient, to determine category membership. The members of a category can be arranged in a typicality gradient, representing the degree to which they represent a typical member of that category. It is also possible for objects to be members of more than one category (e.g., tomatoes as a fruit, or a vegetable).

13.1.3 The Exemplar-based theory

The exemplar-based theory of classification proposes that specific instances, or *exemplars*, act as the prototypes against which other members are compared. Objects are grouped, relative to one another, based on some similarity metric. The exemplar-based theory differs from the prototype theory in that specific instances are the norm against which membership is decided. When asked to name particular members of a category, the attributes of the exemplars are used as cues to retrieve other objects having similar attributes.

13.1.4 The Explanation-based theory

The explanation-based theory of classification proposes that there is an explanation for why categories have the members they do. For instance, the biblical classification of food into *clean* and *unclean* is roughly explained by saying that there should be a correlation between type of habitat, biological structure, and form of locomotion; creatures of the sea should have fins, scales, and swim (sharks and eels don't) and creatures of the land should have four legs (ostriches don't).

From a predictive point of view, explanation-based categories suffer from the problem that they may heavily depend on the knowledge and beliefs of the person who formed the category; for instance, the set of objects a person would remove from their home while it was on fire.

Murphy and Medin[983] discuss how people can use explanations to achieve conceptual coherence in selecting the members of a category (see Table 0.5).

Table 0.5: General properties of explanations and their potential role in understanding conceptual coherence. Adapted from Murphy.[983]

Properties of Explanations	Role in Conceptual Coherence
Explanation of a sort, specified over some domain of observation	Constrains which attributes will be included in a concept representation Focuses on certain relationships over others in detecting attribute correlations
Simplify reality	Concepts may be idealizations that impose more structure than is *objectively* present
Have an external structure— fits in with (or do not contradict) what is already known	Stresses intercategory structure; attributes are considered essential to the degree that they play a part in related theories (external structures)
Have an internal structure— defined in part by relations connecting attributes	Emphasizes mutual constraints among attributes. May suggest how concept attributes are learned
Interact with data and observations in some way	Calls attention to inference processes in categorization and suggests that more than attribute matching is involved

13.2 Measuring similarity

guideline recommendation enforceable

The intent is for these guideline recommendations to be automatically enforceable. This requires an algorithm for calculating similarity, which is the motivation behind the following discussion.

How might two objects be compared for similarity? For simplicity, the following discussion assumes an object can have one of two values for any attribute, yes/no. The discussion is based on material in

Classification and Cognition by W. K. Estes.[398]

similarity product rule

To calculate the similarity of two objects, their corresponding attributes are matched. The product of the similarity coefficient of each of these attributes is computed. A matching similarity coefficient, t (a value in the range one to infinity, and the same for every match), is assigned for matching attributes. A nonmatching similarity coefficient, s_i (a value in the range 0 to 1, and potentially different for each nonmatch), is assigned for each nonmatching coefficient. For example, consider two birds that either have (plus sign), or do not have (minus sign), some attribute (numbered 1 to 6 in Table 0.6). Their similarity, based on these attributes is $t \times t \times s_3 \times t \times s_5 \times t$.

Table 0.6: Computation of pattern similarity. Adapted from Estes.[398]

Attribute	1	2	3	4	5	6
Starling	+	+	-	+	+	+
Sandpiper	+	+	+	+	-	+
Attribute similarity	t	t	s_3	t	s_5	t

When comparing objects within the same category the convention is to give the similarity coefficient, t, for matching attributes, a value of one. Another convention is to give the attributes that differ the same similarity coefficient, s. In the preceding case, the similarity becomes s^2.

Sometimes the similarity coefficient for matches needs to be taken into account. For instance, in the following two examples the similarity between the first two character sequences is ts, while in the second is t^3s. Setting t to be one would result in both pairs of character sequences being considered to have the same similarity, when in fact the second sequence would be judged more similar than the first. Studies on same/different judgments show that both reaction time and error rates increase as a function of the number of items being compared.[777] The value of t cannot always be taken to be unity.

```
A  B          A  B  C  D
A  E          A  E  C  D
```

The previous example computed the similarity of two objects to each other. If we have a category, we can calculate a similarity to category measure. All the members of a category are listed. The similarity of each member, compared with every other member, is calculated in turn and these values are summed for that member. Such a calculation is shown in Table 0.7.

Table 0.7: Computation of similarity to category. Adapted from Estes.[398]

Object	Ro	Bl	Sw	St	Vu	Sa	Ch	Fl	Pe	Similarity to Category
Robin	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Bluebird	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Swallow	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Starling	s	s	s	1	s^3	s^2	s^6	s^5	s^6	$1+3s+s^2+s^3+s^5+2s^6$
Vulture	s^4	s^4	s^4	s^3	1	s^5	s^3	s^2	s^3	$1+s^2+3s^3+3s^4+s^5$
Sandpiper	s	s	s	s^2	s^5	1	s^4	s^5	s^4	$1+3s+s^2+s^4+s^5$
Chicken	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2+s+s^3+s^4+3s^5+s^6$
Flamingo	s^6	s^6	s^6	s^5	s^2	s^5	s	1	s	$1+2s+s^2+2s^5+3s^6$
Penguin	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2+s+s^3+s^4+3s^5+s^6$

Some members of a category are often considered to be more typical of that category than other members. These typical members are sometimes treated as exemplars of that category, and serve as reference points when people are thinking about that category. While there is no absolute measure of typicality, it is possible to compare the typicality of two members of a category. The relative typicality, within a category for two or more objects is calculated from their ratios of similarity to category. For instance, taking the value of s as

0.5, the relative typicality of Robin with respect to Vulture is $4.14/(4.14 + 1.84) = 0.69$, and the relative typicality of Vulture with respect to Robin is $1.84/(4.14 + 1.84) = 0.31$.

It is also possible to create derived categories from existing categories; for instance, large and small birds. For details on how to calculate typicality within those derived categories, see Estes[398] (which also provides experimental results).

similarity contrast model

An alternative measure of similarity is the contrast model. This measure of similarity depends positively on the number of attributes two objects have in common, but negatively on the number of attributes that belong to one but not the other.

$$Contrast\ Sim_{12} = af(F_{12}) - bf(F_1) - cf(F_2) \tag{0.14}$$

where F_{12} is the set of attributes common to objects 1 and 2, F_1 the set of attributes that object 1 has but not object 2, and F_2 the set of attributes that object 2 has but not object 1. The quantities a, b, and c are constants. The function f is some metric based on attributes; the one most commonly appearing in published research is a simple count of attributes.

Taking the example given in Table 0.7, there are four features shared by the starling and sandpiper and one that is unique to each of them. This gives:

$$Contrast\ Sim = 4a - 1b - 1c \tag{0.15}$$

based on bird data we might take, for instance, $a = 1$, $b = 0.5$, and $c = 0.25$ giving a similarity of 3.25.

On the surface, these two models appear to be very different. However, some mathematical manipulation shows that the two methods of calculating similarity are related.

$$Sim_{12} = t^{n_{12}} s^{n_1+n_2} = t^{n_{12}} s^{n_1} s^{n_2} \tag{0.16}$$

Taking the logarithm:

$$\log(Sim_{12}) = n_{12}\log(t) + n_1\log(s) + n_2\log(s) \tag{0.17}$$

letting $a = \log(t)$, $b = \log(s)$, $c = \log(s)$, and noting that the value of s is less than 1, we get:

$$\log(Sim_{12}) = a(n_{12}) - b(n_1) - c(n_2) \tag{0.18}$$

This expression for product similarity has the same form as the expression for contrast similarity. Although b and c have the same value in this example, in a more general form the values of s could be different.

13.2.1 Predicting categorization performance

categorization performance predicting

Studies [1179] have shown that the order in which people list exemplars of categories correlates with their relative typicality ratings. These results lead to the idea that relative typicality ratings could be interpreted as probabilities of categorization responses. However, the algorithm for calculating similarity to category values does not take into account the number of times a subject has encountered a member of the category (which will control the strength of that member's entry in the subject's memory).

For instance, based on the previous example of bird categories when asked to "name the bird which comes most quickly to mind, Robin or Penguin", the probability of Robin being the answer is $4.14/(4.14 + 2.80) = 0.60$, an unrealistically low probability. If the similarity values are weighted according to the frequency

of each member's entry in a subject's memory array (Estes estimated the figures given in Table 0.8), the probability of Robin becomes $1.24/(1.24+0.06) = 0.954$, a much more believable probability. The need to use frequency weightings to calculate a weighted similarity value has been verified by Nosofsky.[1017]

Table 0.8: Computation of weighted similarity to category. From Estes.[398]

Object	Similarity Formula	$s = 0.5$	Relative Frequency	Weighted Similarity
Robin	$3+2s+s^4+2s^5+s^6$	4.14	0.30	1.24
Bluebird	$3+2s+s^4+2s^5+s^6$	4.14	0.20	0.83
Swallow	$3+2s+s^4+2s^5+s^6$	4.14	0.10	0.41
Starling	$1+3s+s^2+s^3+s^5+2s^6$	2.94	0.15	0.44
Vulture	$1+s^2+3s^3+3s^4+s^5$	1.84	0.02	0.04
Sandpiper	$1+3s+s^2+s^4+s^5$	2.94	0.05	0.15
Chicken	$2+s+s^3+s^4+3s^5+s^6$	2.80	0.15	0.42
Flamingo	$1+2s+s^2+2s^5+3s^6$	2.36	0.01	0.02
Penguin	$2+s+s^3+s^4+3s^5+s^6$	2.80	0.02	0.06

The method of measuring similarity just described has been found to be a good predictor of the error probability of people judging which category a stimulus belongs to. The following analysis is based on a study performed by Shepard, Hovland, and Jenkins.[1225]

A simpler example than the bird category is used to illustrate how the calculations are performed. Here, the object attributes are color and shape, made up of the four combinations black/white, triangles/squares. Taking the case where the black triangle and black square have been assigned to category A, and the white triangle and white square have been assigned to category B, we get Table 0.9.

Table 0.9: Similarity to category (black triangle and black square assigned to category A; white triangle and white square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$1+s$	$s+s^2$
Dark square	$1+s$	$s+s^2$
Light triangle	$s+s^2$	$1+s$
Light square	$s+s^2$	$1+s$

If a subject is shown a stimulus that belongs in category A, the expected probability of them assigning it to that category is:

$$\frac{1+s}{(1+s)+(s+s^2)} \Rightarrow \frac{1}{1+s} \tag{0.19}$$

When s is 1 the expected probability is no better than a random choice; when s is 0 the probability is a certainty.

Assigning different stimulus to different categories can change the expected response probability; for instance, by assigning the black triangle and the white square to category A and assigning the white triangle and black square to category B, we get the category similarities shown in Table 0.10.

Table 0.10: Similarity to category (black triangle and white square assigned to category A; white triangle and black square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$s+s^2$	$2s$
Dark square	$2s$	$s+s^2$
Light triangle	$2s$	$s+s^2$
Light square	$s+s^2$	$2s$

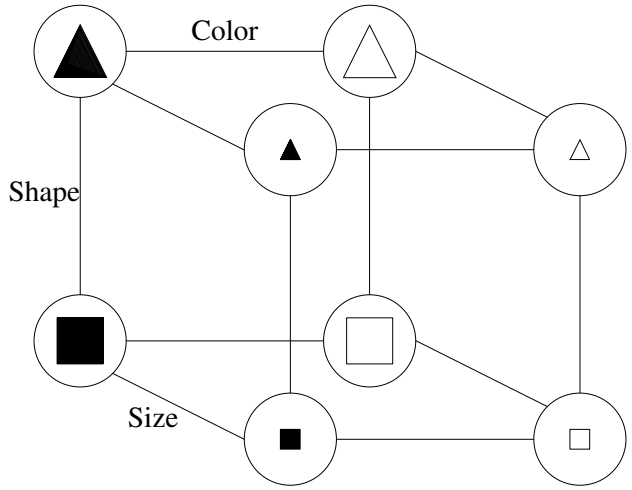


Figure 0.13: Representation of stimuli with shape in the horizontal plane and color in one of the vertical planes. Adapted from Shepard.[1225]

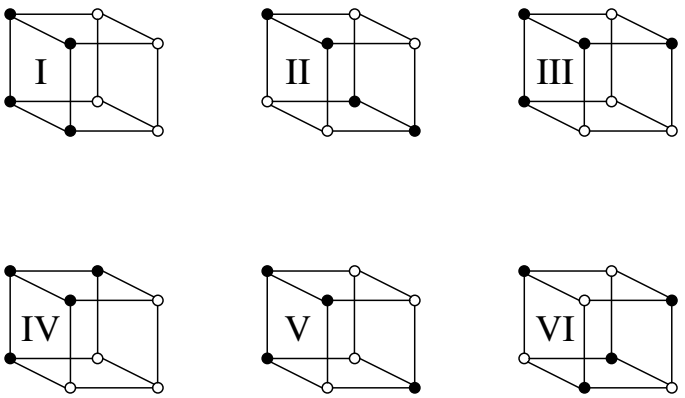


Figure 0.14: One of the six unique configurations (i.e., it is not possible to rotate one configuration into another within the set of six) of selecting four times from eight possibilities. Adapted from Shepard.[1225]

If a subject is shown a stimulus that belongs in category A, the expected probability of them assigning it to that category is:

$$\frac{1+s^2}{(2s)+(1+s^2)} \Rightarrow \frac{1+s^2}{(1+s)^2} \tag{0.20}$$

For all values of s between 0 and 1 (but not those two values), the probability of a subject assigning a stimulus to the correct category is always less than for the category defined previously, in this case.

In the actual study performed by Shepard, Hovland, and Jenkins,[1225] stimuli that had three attributes, color/size/shape, were used. If there are two possible values for each of these attributes, there are eight possible stimuli (see Figure 0.13).

Each category was assigned four different members. There are 70 different ways of taking four things from a choice of eight $(8!/(4!4!))$, creating 70 possible categories. However, many of these 70 different categories share a common pattern; for instance, all having one attribute, like all black or all triangles. If this symmetry is taken into account, there are only six different kinds of categories. One such selection of six

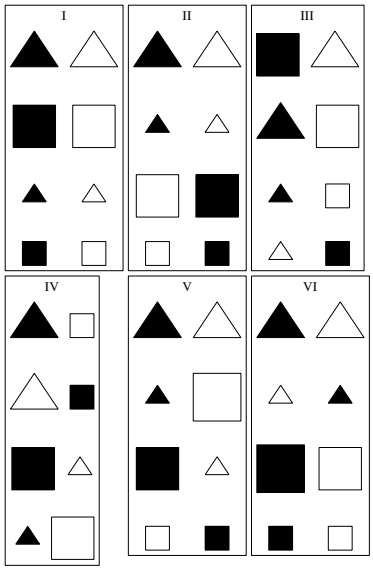


Figure 0.15: Example list of categories. Adapted from Shepard.[1225]

categories is shown in Figure 0.14, the black circles denoting the selected attributes.

Having created these six categories, Shepard et al. trained and measured the performance of subjects in assigning presented stimuli (one of the list of 70 possible combinations of four things— Figure 0.15) to one of them.

Estes[398] found a reasonable level of agreement between the error rates reported by Shepard et al. and the rates predicted by the similarity to category equations. There is also a connection between categorization performance and Boolean complexity; this is discussed elsewhere.

1739 selection statement
syntax

A series of studies by Feldman[414] was able to show a correlation between the difficulty subjects had answering the Shepard classification problems and their boolean complexity (i.e., the length of the shortest logically equivalent propositional formula).

13.3 Cultural background and use of information

The attributes used to organize information (e.g., categorize objects) has been found to vary across cultures[1015] and between experts and non-experts. The following studies illustrate how different groups of people agree or differ in their categorization behavior (a cultural difference in the naming of objects is discussed elsewhere):

categorization
cultural differences

792 naming
cultural differences

- A study by Bailenson, Shum, and Coley[81] asked US bird experts (average of 22.4 years bird watching), US undergraduates, and ordinary Itzaj (Maya Amerindians people from Guatemala) to sort two sets (of US and Maya) of 104 bird species into categories. The results found that the categorization choices made by the three groups of subjects were internally consistent within each group. The correlation between the taxonomies, created by the categories, and a published scientific taxonomy of US experts (0.60 US birds, 0.70 Maya birds), Itzaj (0.45, 0.61), and nonexperts (0.38, 0.34). The US experts correlated highly with the scientific taxonomy for both sets of birds, the Itzaj only correlated highly for Maya birds, and the nonexperts had a low correlation for either set of birds. The reasons given for the Maya choices varied between the expert groups; US experts were based on a scientific taxonomy, Itzaj were based on ecological justifications (the birds relationship with its environment). Cultural differences were found in that, for instance, US subjects were more likely to generalise from songbirds, while the Itzaj were more likely to generalize from perceptually striking birds.

- A study by Proffitt, Coley, and Medin[1126] told three kinds of tree experts (landscape gardeners, parks maintenance workers, scientists researching trees) about a new disease that affected two kinds of tree (e.g., Horsechestnut and Ohio buckeye). Subjects were then asked what other trees they thought might also be affected by this disease. The results showed differences between kinds of experts in the kinds of justifications given for the answers. For instance, landscapers and maintenance workers used more causal/ecological explanations (tree distribution, mechanism of disease transmission, resistance, and susceptibility) and fewer similarity-based justifications (species diversity and family size). For taxonomists this pattern was reversed.

14 Decision making

Writing source code is not a repetitive process. Developers have to think about what they are going to write, which means they have to make decisions. Achieving the stated intent of these coding guidelines (minimizing the cost of ownership source code) requires that they be included in this, developer, decision-making process.

coding 0 guidelines introduction

There has been a great deal of research into how and why people make decisions in various contexts. For instance, consumer research trying to predict how a shopper will decide among packets of soap powder on a supermarket shelf. While the items being compared and their attributes vary (e.g., which soap will wash the whitest, should an **`if`** statement or a **`switch`** statement be used; which computer is best), the same underlying set of mechanisms appear to be used, by people, in making decisions.

The discussion in this section has been strongly influenced by *The Adaptive Decision Maker* by Payne, Bettman, and Johnson.[1065] The model of human decision making proposed by Payne et al. is based on the idea that people balance the predicted cognitive effort required to use a particular decision-making strategy against the likely accuracy achieved by that decision-making strategy. The book lists the following major assumptions:

Payne[1065]

- *Decision strategies are sequences of mental operations that can be usefully represented as productions of the form IF (condition 1, . . . , condition n) THEN (action 1, . . . , action m).*
- *The cognitive effort needed to reach a decision using a particular strategy is a function of the number and type of operators (productions) used by that strategy, with the relative effort levels of various strategies contingent on task environments.*
- *Different strategies are characterized by different levels of accuracy, with the relative accuracy levels of various strategies contingent on task environments.*
- *As a result of prior experiences and training, a decision maker is assumed to have more than one strategy (sequence of operations) available to solve a decision problem of any complexity.*
- *Individuals decide how to decide primarily by considering both the cognitive effort and the accuracy of the various strategies.*
- *Additional considerations, such as the need to justify a decision to others or the need to minimize the conflict inherent in a decision problem, may also impact strategy selection.*
- *The decision of how to decide is sometimes a conscious choice and sometimes a learned contingency among elements of the task and the relative effort and accuracy of decision strategies.*
- *Strategy selection is generally adaptive and intelligent, if not optimal.*

14.1 Decision-making strategies

Before a decision can be made it is necessary to select a decision-making strategy. For instance, a developer who is given an hour to write a program knows there is insufficient time for making complicated trade-offs among alternatives. When a choice needs to be made, the likely decision-making strategy adopted would be to compare the values of a single attribute, the estimated time required to write the code (a decision-making strategy based on looking at the characteristics of a single attribute is known as the lexicographic heuristic).

lexicographic 0 heuristic decision making

Researchers have found that people use a number of different decision-making strategies. In this section we discuss some of these strategies and the circumstances under which people might apply them. The

list of strategies discussed in the following subsections is not exhaustive, but it does cover many of the decision-making strategies used when writing software.

The strategies differ in several ways. For instance, some make trade-offs among the attributes of the alternatives (making it possible for an alternative with several good attributes to be selected instead of the alternative whose only worthwhile attribute is excellent), while others make no such trade-offs. From the human perspective, they also differ in the amount of information that needs to be obtained and the amount of (brain) processing needed to make a decision. A theoretical analysis of the cost of decision making is given by Shugan.[1231]

14.1.1 The weighted additive rule

weighted additive rule

The weighted additive rule requires the greatest amount of effort, but delivers the most accurate result. It also requires that any conflicts among different attributes be confronted. Confronting conflict is something, as we shall see later, that people do not like doing. This rule consists of the following steps:

1. Build a list of attributes for each alternative.
2. Assign a value to each of these attributes.
3. Assign a weight to each of these attributes (these weights could, for instance, reflect the relative importance of that attribute to the person making the decision, or the probability of that attribute occurring).
4. For each alternative, sum the product of each of its attributes' value and weight.
5. Select the alternative with the highest sum.

An example, where this rule might be applied, is in deciding whether an equality test against zero should be made before the division of two numbers inside a loop. Attributes might include performance and reliability. If a comparison against zero is made the performance will be decreased by some amount. This disadvantage will be given a high or low weight depending on whether the loop is time-critical or not. The advantage is that reliability will be increased because the possibility of a divide by zero can be avoided. If a comparison against zero is not made, there is no performance penalty, but the reliability could be affected (it is necessary to take into account the probability of the second operand to the divide being zero).

14.1.2 The equal weight heuristic

The equal weight heuristic is a simplification of the weighted additive rule in that it assigns the same weight to every attribute. This heuristic might be applied when accurate information on the importance of each attribute is not available, or when a decision to use equal weights has been made.

14.1.3 The frequency of good and bad features heuristic

People do not always have an evaluation function for obtaining the value of an attribute. A simple estimate in terms of good/bad is sometimes all that is calculated (looking at things in black and white). By reducing the range of attribute values, this heuristic is a simplification of the equal weight heuristic, which in turn is a simplification of the weighted additive rule. This rule consists of the following steps:

1. List the good and bad attributes of every alternative.
2. Calculate the sum of each attributes good and the sum of its bad attributes.
3. Select the alternative with the highest count of either good or bad attributes, or some combination of the two.

A coding context, where a good/bad selection might be applicable, occurs in choosing the type of an object. If the object needs to hold a fractional part, it is tempting to use a floating type rather than an integer type (perhaps using some scaling factor to enable the fractional part to be represented). Drawing up a list of good and bad attributes ought to be relatively straight-forward; balancing them, to select a final type, might be a little more contentious

14.1.4 The majority of confirming dimensions heuristic

While people may not be able to explicitly state an evaluation function that provides a numerical measure of an attribute, they can often give a yes/no answer to the question: *Is the value of attribute X greater (or less) for alternative A compared to alternative B?*. This enables them to determine which alternative has the most (or least) of each attribute. This rule consists of the following steps:

1. Select a pair of alternatives.
2. Compare each matching attribute in the two alternatives.
3. Select the alternative that has the greater number of winning attributes.
4. Pair the winning alternative with an uncompared alternative and repeat the compare/select steps.
5. Once all alternatives have been compared at least once, the final winning alternative is selected.

In many coding situations there are often only two viable alternatives. Pairwise comparison of their attributes could be relatively easy to perform. For instance, when deciding whether to use a sequence of **`if`** statements or a **`switch`** statement, possible comparison attributes include efficiency of execution, readability, ease of changeability (adding new cases, deleting, or merging existing ones).

14.1.5 The satisficing heuristic

satisficing heuristic decision making

The result of the satisficing heuristic depends on the order in which alternatives are checked and often does not check all alternatives. Such a decision strategy, when described in this way, sounds unusual, but it is simple to perform. This rule consists of the following steps:

1. Assign a cutoff, or aspirational, level that must be met by each attribute.
2. Perform the following for each alternative:
   - Check each of its attributes against the cutoff level, rejecting the alternative if the attribute is below the cutoff.
   - If there are no attributes below the cutoff value, accept this alternative.
3. If no alternative is accepted, revise the cutoff levels associated the attributes and repeat the previous step.

An example of the satisficing heuristic might be seen when selecting a library function to return some information to a program. The list of attributes might include the amount of information returned and the format it is returned in (relative to the format it is required to be in). Once a library function meeting the developer's minimum aspirational level has been found, additional effort is not usually invested in finding a better alternative.

14.1.6 The lexicographic heuristic

lexicographic heuristic decision making

The lexicographic heuristic has a low effort cost, but it might not be very accurate. It can also be intransitive; with X preferred to Y, Y preferred to Z, and Z preferred to X. This rule consists of the following steps:

1. Determine the most important attribute of all the alternatives.
2. Find the alternative that has the best value for the selected most important attribute.
3. If two or more alternatives have the same value, select the next most important attribute and repeat the previous step using the set of alternatives whose attribute values tied.
4. The result is the alternative having the best value on the final, most important, attribute selected.

An example of the intransitivity that can occur, when using this heuristic, might be seen when writing software for embedded applications. Here the code has to fit within storage units that occur in fixed-size increments (e.g., 8 K chips). It may be possible to increase the speed of execution of an application by writing code for specific special cases; or have generalized code that is more compact, but slower. We might have the following, commonly seen, alternatives (see Table 0.11).

Table 0.11: Storage/Execution performance alternatives.

Alternative	Storage Needed	Speed of Execution
X	7 K	Low
Y	15 K	High
Z	10 K	Medium

Based on storage needed, X is preferred to Y. But because storage comes in 8 K increments there is no preference, based on this attribute, between Y and Z; however, Y is preferred to Z based on speed of execution. Based on speed of execution Z is preferred to X.

14.1.6.1 The elimination-by-aspects heuristic

The elimination-by-aspects heuristic uses cutoff levels, but it differs from the satisficing heuristic in that alternatives are eliminated because their attributes fall below these levels. This rule consists of the following steps:

1. The attributes for all alternatives are ordered (this ordering might be based on some weighting scheme).
2. For each attribute in turn, starting with the most important, until one alternative remains:
   - Select a cutoff value for that attribute.
   - Eliminate all alternatives whose value for that attribute is below the cutoff.
3. Select the alternative that remains.

This heuristic is often used when there are resource limitations, for instance, deadlines to meet, performance levels to achieve, or storage capacities to fit within.

14.1.7 The habitual heuristic

The habitual heuristic looks for a match of the current situation against past situations, it does not contain any evaluation function (although there are related heuristics that evaluate the outcome of previous decisions). This rule consists of the step:

1. select the alternative chosen last time for that situation.

Your author's unsubstantiated claim is that this is the primary decision-making strategy used by software developers.

Sticking with a winning solution suggests one of two situations:

1. So little is known that once a winning solution is found, it is better to stick with it than to pay the cost (time and the possibility of failure) of looking for a better solution that might not exist.
2. The developer has extensively analyzed the situation and knows the best solution.

Coding decisions are not usually of critical importance. There are many solutions that will do a satisfactory job. It may also be very difficult to measure the effectiveness of any decision, because there is a significant delay between the decision being made and being able to measure its effect. In many cases, it is almost

impossible to separate out the effect of one decision from the effects of all the other decisions made (there may be a few large coding decisions, but the majority are small ones).

recognition-primed decision making

A study by Klein[747] describes how fireground commanders use their experience to size-up a situation very rapidly. Orders are given to the firefighters under their command without any apparent decisions being made (in their interviews they even found a fireground commander who claimed that neither he, nor other commanders, ever made decisions; they knew what to do). Klein calls this strategy *recognition-primed decision making*.

14.2 Selecting a strategy

Although researchers have uncovered several decision-making strategies that people use, their existence does not imply that people will make use of all of them. The strategies available to individuals can vary depending on their education, training, and experience. A distinction also needs to be made between a person's knowledge of a strategy (through education and training) and their ability to successfully apply it (perhaps based on experience).

The task itself (that creates the need for a decision to be made) can affect the strategy used. These task effects include task complexity, the response mode (how the answer needs to be given), how the information is displayed, and context. The following subsections briefly outline these effects.

14.2.1 Task complexity

task complexity decision making

In general the more complex the decision, the more people will tend to use simplifying heuristics. The following factors influence complexity:

- *Number of alternatives.* As the number of alternatives that need to be considered grows, there are shifts in the decision-making strategy used.
- *Number of attributes.* Increasing the number of attributes increases the confidence of people's judgments, but it also increases their variability. The evidence for changes in the quality of decision making, as the number of attributes increases, is less clear-cut. Some studies show a decrease in quality; it has been suggested that people become overloaded with information. There is also the problem of deciding what constitutes a high-quality decision.
- *Time pressure.* People have been found to respond to time pressure in one of several ways. Some respond by accelerating their processing of information, others respond by reducing the amount of information they process (by filtering the less important information, or by concentrating on certain kinds of information such as negative information), while others respond by reducing the range of ideas and actions considered.

14.2.2 Response mode

There are several different response modes. For instance, a choice response mode frames the alternatives in terms of different choices; a matching response mode presents a list of questions and answers and the decision maker has to provide a matching answer to a question; a bidding response mode requires a value to be given for buying or selling some object. There are also other response modes, that are not listed here.

The choice of response mode, in some cases, has been shown to significantly influence the preferred alternatives. In extreme cases, making a decision may result in X being preferred to Y, while the mathematically equivalent decision, presented using a different response mode, can result in Y being preferred to X. For instance, in gambling situations it has been found that people will prefer X to Y when asked to select between two gambles (where X has a higher probability of winning, but with lower amounts), but when asked to bid on gambles they prefer Y to X (with Y representing a lower probability of winning a larger amount).

Such behavior breaks what was once claimed to be a fundamental principle of rational decision theory, *procedure invariance*. The idea behind this principle was that people had an invariant (relatively) set of internal preferences that were used to make decisions. These experiments showed that sometimes preferences

are constructed on the fly. Observed preferences are likely to take a person's internal preferences and the heuristics used to construct the answer into account.

Code maintenance is one situation where the task can have a large impact on how the answer is selected. When small changes are made to existing code, many developers tend to operate in a matching mode, choosing constructs similar, if not identical, to the ones in the immediately surrounding lines of code. If writing the same code from scratch, there is nothing to match, another response mode will necessarily need to be used in deciding what constructs to use.

A lot of the theoretical discussion on the reasons for these response mode effects has involved distinguishing between judgment and choice. People can behave differently, depending on whether they are asked to make a judgment or a choice. When writing code, the difference between judgment and choice is not always clear-cut. Developers may believe they are making a choice between two constructs when in fact they have already made a judgment that has reduced the number of alternatives to choose between.

Writing code is open-ended in the sense that theoretically there are an infinite number of different ways of implementing what needs to be done. Only half a dozen of these might be considered sensible ways of implementing some given functionality, with perhaps one or two being commonly used. Developers often limit the number of alternatives under consideration because of what they perceive to be overriding external factors, such as preferring an inline solution rather than calling a library function because of alleged quality problems with that library. One possibility is that decision making during coding be considered as a two-stage process, using judgment to select the alternatives, from which one is chosen.

14.2.3 Information display

Studies have shown that how information, used in making a decision, is displayed can influence the choice of a decision-making strategy.[1200] These issues include: only using the information that is visible (the concreteness principle), the difference between available information and processable information (displaying the price of one brand of soap in dollars per ounce, while another brand displays francs per kilogram), the completeness of the information (people seem to weigh common attributes more heavily than unique ones, perhaps because of the cognitive ease of comparison), and the format of the information (e.g., digits or words for numeric values).

What kind of information is on display when code is being written? A screen's worth of existing code is visible on the display in front of the developer. There may be some notes to the side of the display. All other information that is used exists in the developer's head.

Existing code is the result of past decisions made by the developer; it may also be modified by future decisions that need to be made (because of a need to modify the behavior of this existing code). For instance, the case in which another conditional statement needs to be added within a deeply nested series of conditionals. The information display (layout) of the existing code can affect the developer's decision about how the code is to be modified (a function, or macro, might be created instead of simply inserting the new conditional). Here the information display itself is an attribute of the decision making (code wrapping, at the end of a line, is an attribute that has a yes/no answer).

14.2.4 Agenda effects

agenda effects
decision making

The agenda effect occurs when the order in which alternatives are considered influences the final answer. For instance, take alternatives X, Y, and Z and group them into some form of hierarchy before performing a selection. When asked to choose between the pair [X, Y] and Z (followed by a choice between X and Y if that pair is chosen) and asked to choose between the pair [X, Z] and Y (again followed by another choice if that pair is chosen), an agenda effect would occur if the two final answers were different.

An example of the agenda effect is the following. When writing coding, it is sometimes necessary to decide between writing in line code, using a macro, or using a function. These three alternatives can be grouped into a natural hierarchy depending on the requirements. If efficiency is a primary concern, the first decision may be between `[in line, macro]` and `function`, followed by a decision between `in line` and `macro` (if that pair is chosen). If we are more interested in having some degree of abstraction, the first decision is likely to be between `[macro, function]` and `in line` (see Figure 0.16).

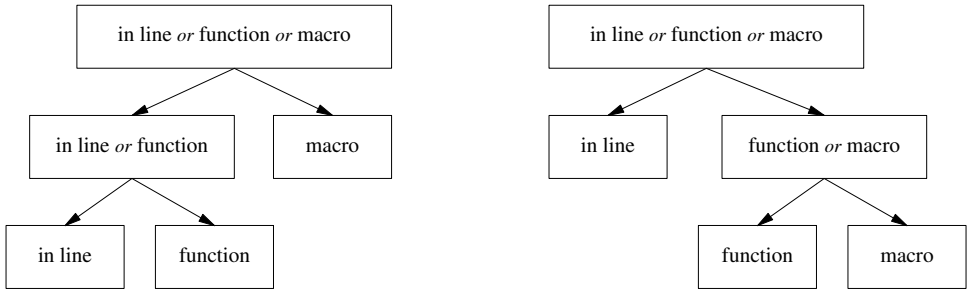


Figure 0.16: Possible decision paths when making pair-wise comparisons on whether to use a inline code, a function, or a macro; for two different pair-wise associations.

In the efficiency case, if performance is important in the context of the decision, `[in line, macro]` is likely to be selected in preference to `function`. Once this initial choice has been made other attributes can be considered (since both alternatives have the same efficiency). We can now decide whether abstraction is considered important enough to select `macro` over `in line`.

If the initial choice had been between `[macro, function]` and `in line`, the importance of efficiency would have resulted in `in line` being chosen (when paired with `function`, `macro` appears less efficient by association).

14.2.5 Matching and choosing

When asked to make a decision based on *matching*, a person is required to specify the value of some variable such that two alternatives are considered to be equivalent. For instance, how much time should be spent testing 200 lines of code to make it as reliable as the 500 lines of code that has had 10 hours of testing invested in it? When asked to make a decision based on *choice*, a person is presented with a set of alternatives and is required to specify one of them.

A study by Tversky, Sattath, and Slovic[1376] investigated the *prominence hypothesis*. This proposes that when asked to make a decision based on choice, people tend to use the prominent attributes of the options presented (adjusting unweighted intervals being preferred for matching options). Their study suggested that there were differences between the mechanisms used to make decisions for matching and choosing.

14.3 The developer as decision maker

The writing of source code would seem to require developers to make a very large number of decisions. However, experience shows that developers do not appear to be consciously making many decisions concerning what code to write. Most decisions being made involve issues related to the mapping from the application domain, choosing algorithms, and general organizational issues (i.e., where functions or objects should be defined).

Many of the coding-level decisions that need to be made occur again and again. Within a year or so, in full-time software development, sufficient experience has usually been gained for many decisions to be reduced to matching situations against those previously seen, and selecting the corresponding solution. For instance, the decision to use a series of **`if`** statements or a **`switch`** statement might require the pattern *same variable tested against integer constant and more than two tests are made* to be true before a **`switch`** statement is used. This is what Klein[747] calls recognition-primed decision making. This code writing methodology works because there is rarely a need to select the optimum alternative from those available.

recognition-primed decision making 0

Some decisions occur to the developer as code is being written. For instance, a developer may notice that the same sequence of statements, currently being written, was written earlier in a different part of the source (or perhaps it will occur to the developer that the same sequence of statements is likely to be needed in code that is yet to be written). At this point the developer has to make a decision about making a decision (metacognition). Should the decision about whether to create a function be put off until the current work item is completed, or should the developer stop what they are currently doing to make a decision on whether to

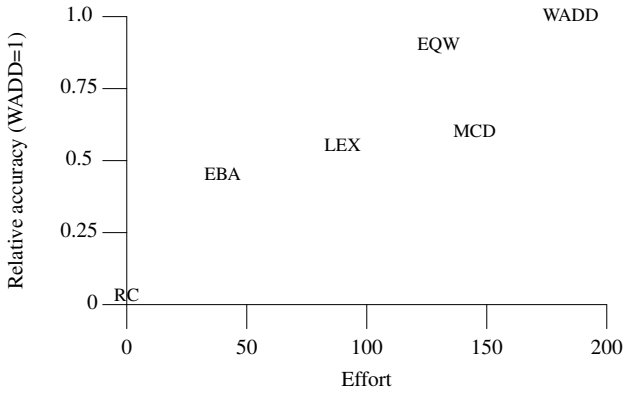


Figure 0.17: Effort and accuracy levels for various decision-making strategies; EBA (Elimination-by-aspects heuristic), EQW (equal weight heuristic), LEX (lexicographic heuristic), MCD (majority of confirming dimensions heuristic), RC (Random choice), and WADD (weighted additive rule). Adapted from Payne.[1065]

turn the statement sequence into a function definition? Remembering work items and metacognitive decision processes are handled by a developer's attention. The subject of *attention* is discussed elsewhere. 0 attention

Just because developers are not making frequent, conscious decisions does not mean that their choices are consistent and repeatable (they will always make the same decision). There are a number of both internal and external factors that may affect the decisions made. Researchers have uncovered a wide range of issues, a few of which are discussed in the following subsections.

14.3.1 Cognitive effort vs. accuracy

People like to make accurate decisions with the minimum of effort. In practice, selecting a decision-making strategy requires trading accuracy against effort (or to be exact, expected effort making the decision; the actual effort required can only be known after the decision has been made). effort vs. accuracy decision making

The fact that people do make effort/accuracy trade-offs is shown by the results from a wide range of studies (this issue is also discussed elsewhere, and Payne et al.[1065] discuss this topic in detail). See Figure 0.17 for a comparison. 0 cost/accuracy trade-off

The extent to which any significant cognitive effort is expended in decision making while writing code is open to debate. A developer may be expending a lot of effort on thinking, but this could be related to problem solving, algorithmic, or design issues.

One way of performing an activity that is not much talked about, is *flow*— performing an activity without any conscious effort— often giving pleasure to the performer. A best-selling book on the subject of flow[302] is subtitled "The psychology of optimal experience", something that artistic performers often talk about. Developers sometimes talk of *going with the flow* or *just letting the writing flow* when writing code; something writers working in any medium might appreciate. However, it is your author's experience that this method of working often occurs when deadlines approach and developers are faced with writing a lot of code quickly. Code written using *flow* is often very much like a river; it has a start and an ending, but between those points it follows the path of least resistance, and at any point readers rarely have any idea of where it has been or where it is going. While works of fiction may gain from being written in this way, the source code addressed by this book is not intended to be read for enjoyment. While developers may enjoy spending time solving mysteries, their employers do not want to pay them to have to do so. developer flow

Code written using *flow* is not recommended, and is not discussed further here. The use of intuition is discussed elsewhere. 0 developer intuition

14.3.2 Which attributes are considered important?

Developers tend to consider mainly technical attributes when making decisions. Economic attributes are often ignored, or considered unimportant. No discussion about attributes would be complete without mentioning developer fun

fun. Developers have gotten used to the idea that they can enjoy themselves at work, doing *fun* things. Alternatives that have a negative value for the fun attribute, and a large positive value for the time to carry out attribute are often quickly eliminated.

The influence of developer enjoyment on decision making, can be seen in many developers' preference for writing code, rather than calling a library function. On a larger scale, the often-heard developer recommendation for rewriting a program, rather than reengineering an existing one, is motivated more by the expected pleasure of writing code than the economics (and frustration) of reengineering.

One reason for the lack of consideration of economic factors is that many developers have no training, or experience in this area. Providing training is one way of introducing an economic element into the attributes used by developers in their decision making.

14.3.3 Emotional factors

developer emotional factors

weighted additive rule

Many people do not like being in a state of conflict and try to avoid it. Making a decision can create conflict, by requiring one attribute to be traded off against another. For instance, having to decide whether it is more important for a piece of code to execute quickly or reliably. It has been argued that people will avoid strategies that involve difficult, emotional, value trade-offs.

Emotional factors relating to source code need not be limited to internal, private developer decision making. During the development of an application involving more than one developer, particular parts of the source are often considered to be *owned* by an individual developer. A developer asked to work on another developers source code, perhaps because that person is away, will sometimes feel the need to adopt the *style* of that developer, making changes to the code in a way that is thought to be acceptable to the absent developer. Another approach is to ensure that the changes stand out from the *owner's* code. On the owning developer's return, the way in which changes were made is explained. Because they stand out, developers can easily see what changes were made to *their* code and decide what to do about them.

People do not like to be seen to make mistakes. It has been proposed[383] that people have difficulty using a decision-making strategy, that makes it explicit that there is some amount of error in the selected alternative. This behavior occurs even when it can be shown that the strategy would lead to better, on average, solutions than the other strategies available.

14.3.4 Overconfidence

overconfidence

A person is overconfident when their belief in a proposition is greater than is warranted by the information available to them. It has been argued that overconfidence is a useful attribute that has been selected for by evolution. Individuals who overestimates their ability are more likely to undertake activities they would not otherwise have been willing to do. Taylor and Brown[1332] argue that a theory of mental health defined in terms of contact with reality does not itself have contact with reality: "Rather, the mentally healthy person appears to have the enviable capacity to distort reality in a direction that enhances self-esteem, maintains beliefs in personal efficacy, and promotes an optimistic view of the future."

confirmation bias

Numerous studies have shown that most people are overconfident about their own abilities compared with others. People can be overconfident in their ability for several reasons: confirmation bias can lead to available information being incorrectly interpreted; a person's inexpert calibration (the degree of correlation between confidence and performance) of their own abilities is another reason. A recent study[746] has also highlighted the importance of the how, what, and whom of questioning in overconfidence studies. In some cases, it has been shown to be possible to make overconfidence disappear, depending on how the question is asked, or on what question is asked. Some results also show that there are consistent individual differences in the degree of overconfidence.

Charles Darwin. In *The descent of man*, 1871, p. 3

ignorance more frequently begets confidence than does knowledge

A study by Glenberg and Epstein[497] showed the danger of a little knowledge. They asked students, who were studying either physics or music, to read a paragraph illustrating some central principle (of physics or music). Subjects were asked to rate their confidence in being able to accurately answer a question about

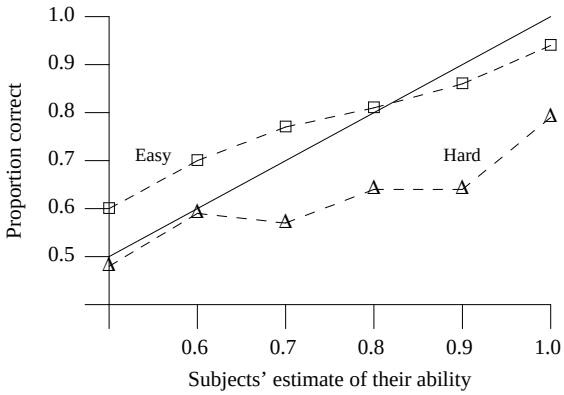


Figure 0.18: Subjects' estimate of their ability (bottom scale) to correctly answer a question and actual performance in answering on the left scale. The responses of a person with perfect self-knowledge is given by the solid line. Adapted from Lichtenstein.[855]

the text. They were then presented with a statement drawing some conclusion about the text (it was either true or false), which they had to answer. They then had to rate their confidence that they had answered the question correctly. This process was repeated for a second statement, which differed from the first in having the opposite true/false status.

The results showed that the more physics or music courses a subject had taken, the more confident they were about their own abilities. However, a subject's greater confidence in being able to correctly answer a question, before seeing it, was not matched by a greater ability to provide the correct answer. In fact as subjects' confidence increased, the accuracy of the calibration of their own ability went down. Once they had seen the question, and answered it, subjects were able to accurately calibrate their performance.

Subjects did not learn from their previous performances (in answering questions). They could have used information on the discrepancy between their confidence levels before/after seeing previous questions to improve the accuracy of their confidence estimates on subsequent questions.

The conclusion drawn by Glenberg and Epstein was that subjects' overconfidence judgments were based on self-classification as an expert, within a domain, not the degree to which they comprehended the text.

A study by Lichtenstein and Fishhoff[855] discovered a different kind of overconfidence effect. As the difficulty of a task increased, the accuracy of people's estimates of their own ability to perform the task decreased. In this study subjects were asked general knowledge questions, with the questions divided into two groups, hard and easy. The results in Figure 0.18 show that subjects' overestimated their ability (bottom scale) to correctly answer (actual performance, left scale) hard questions. On the other hand, they underestimated their ability to answer easy questions. The responses of a person with perfect self-knowledge are given by the solid line.

These, and subsequent results, show that the skills and knowledge that constitute competence in a particular domain are the same skills needed to evaluate one's (and other people's) competence in that domain. People who do not have these skills and knowledge lack metacognition (the name given by cognitive psychologists to the ability of a person to accurately judge how well they are performing). In other words, the knowledge that underlies the ability to produce correct judgment is the same knowledge that underlies the ability to recognize correct judgment.

Some very worrying results, about what overconfident people will do, were obtained in a study performed by Arkes, Dawes, and Christensen.[54] This study found that subjects used a formula that calculated the best decision in a probabilistic context (provided to them as part of the experiment) less when incentives were provided or the subjects thought they had domain expertise. This behavior even continued when the subjects were given feedback on the accuracy of their own decisions. The explanation, given by Arkes et al., was that when incentives were provided, people changed decision-making strategies in an attempt to beat the odds. Langer[809] calls this behavior *the illusion of control*.

Developers overconfidence and their aversion to explicit errors can sometimes be seen in the handling of floating-point calculations. A significant amount of mathematical work has been devoted to discovering the bounds on the errors for various numerical algorithms. Sometimes it has been proved that the error in the result of a particular algorithm is the minimum error attainable (there is no algorithm whose result has less error). This does not seem to prevent some developers from believing that they can design a more accurate algorithm. Phrases, such as *mean error* and *average error*, in the presentation of an algorithm's error analysis do not help. An overconfident developer could take this as a hint that it is possible to do better for the conditions that prevail in his (or her) application (and not having an error analysis does not disprove it is not better).

14.4 The impact of guideline recommendations on decision making

A set of guidelines can be more than a list of recommendations that provide a precomputed decision matrix. A guidelines document can provide background information. Before making any recommendations, the author(s) of a guidelines document need to consider the construct in detail. A good set of guidelines will document these considerations. This documentation provides a knowledge base of the alternatives that might be considered, and a list of the attributes that need to be taken into account. Ideally, precomputed values and weights for each attribute would also be provided. At the time of this writing your author only has a vague idea about how these values and weights might be computed, and does not have the raw data needed to compute them.

justifying 0 decisions

A set of guideline recommendations can act as a lightening rod for decisions that contain an emotional dimension. Adhering to coding guidelines being the justification for the decision that needs to be made. Having to justify decisions can affect the decision-making strategy used. If developers are expected to adhere to a set of guidelines, the decisions they make could vary depending on whether the code they write is independently checked (during code review, or with a static analysis tool).

14.5 Management's impact on developers' decision making

Although lip service is generally paid to the idea that coding guidelines are beneficial, all developers seem to have heard of a case where having to follow guidelines has been counterproductive. In practice, when first introduced, guidelines are often judged by both the amount of additional work they create for developers and the number of faults they immediately help locate. While an automated tool may uncover faults in existing code, this is not the primary intended purpose of using these coding guidelines. The cost of adhering to guidelines in the present is paid by developers; the benefit is reaped in the future by the owners of the software. Unless management successfully deals with this cost/benefit situation, developers could decide it is not worth their while to adhere to guideline recommendations.

What factors, controlled by management, have an effect on developers' decision making? The following subsections discuss some of them.

14.5.1 Effects of incentives

Some deadlines are sufficiently important that developers are offered incentives to meet them. Studies, on use of incentives, show that their effect seems to be to make people work harder, not necessarily smarter.

Increased effort is thought to lead to improved results. Research by Paese and Sniezek[1041] found that increased effort led to increased confidence in the result, but without there being any associated increase in decision accuracy.

Before incentives can lead to a change of decision-making strategies, several conditions need to be met:

- The developer must believe that a more accurate strategy is required. Feedback on the accuracy of decisions is the first step in highlighting the need for a different strategy,[582] but it need not be sufficient to cause a change of strategy.
- A better strategy must be available. The information needed to be able to use alternative strategies may not be available (for instance, a list of attribute values and weights for a weighted average strategy).
- The developer must believe that they are capable of performing the strategy.

14.5.2 Effects of time pressure

Research by Payne, Bettman, and Johnson,[1065] and others, has shown that there is a hierarchy of responses for how people deal with time pressure:

1. They work faster.
2. If that fails, they may focus on a subset of the issues.
3. If that fails, they may change strategies (e.g., from alternative based to attribute based).

If the time pressure is on delivering a finished program, and testing has uncovered a fault that requires changes to the code, then the weighting assigned to attributes is likely to be different than during initial development. For instance, the risk of a particular code change impacting other parts of the program is likely to be a highly weighted attribute, while maintainability issues are usually given a lower weighting as deadlines approach.

14.5.3 Effects of decision importance

Studies investigating at how people select decision-making strategies have found that increasing the benefit for making a correct decision, or having to make a decision that is irreversible, influences how rigorously a strategy is applied, not which strategy is applied.[105]

The same coding construct can have a different perceived importance in different contexts. For instance, defining an object at file scope is often considered to be a more important decision than defining one in block scope. The file scope declaration has more future consequences than the one in block scope.

An irreversible decision might be one that selects the parameter ordering in the declaration of a library function. Once other developers have included calls to this function in their code, it can be almost impossible (high cost/low benefit) to change the parameter ordering.

14.5.4 Effects of training

A developer's training in software development is often done using examples. Sample programs are used to demonstrate the solutions to small problems. As well as learning how different constructs behave, and how they can be joined together to create programs, developers also learn what attributes are considered to be important in source code. They learn the implicit information that is not written down in the text books. Sources of implicit learning include the following:

- *The translator used for writing class exercises.* All translators have their idiosyncrasies and beginners are not sophisticated enough to distinguish these from truly generic behavior. A developer's first translator usually colors his view of writing code for several years.
- *Personal experiences during the first few months of training.* There are usually several different alternatives for performing some operation. A bad experience (perhaps being unable to get a program that used a block scope array to work, but when the array was moved to file scope the program worked) with some construct can lead to a belief that use of that construct was problem-prone and to be avoided (all array objects being declared, by that developer, at file scope and never in block scope).
- *Instructor biases.* The person teaching a class and marking submitted solutions will impart their own views on what attributes are important. Efficiency of execution is an attribute that is often considered to be important. Its actual importance, in most cases, has declined from being crucial 50 years ago to being almost a nonissue today. There is also the technical interest factor in trying to write code as efficiently as possible. A related attribute is program size. Praise is more often given for short programs, rather than longer ones. There are applications where the size of the code is important, but generally time spent writing the shortest program is wasted (and may even be more difficult to comprehend than a longer program).
- *Consideration for other developers.* Developers are rarely given practical training on how to read code, or how to write code that can easily be read by others. Developers generally believe that any difficulty others experience in comprehending their code is not caused by how they wrote it.

- *Preexisting behavior.* Developers bring their existing beliefs and modes of working to writing C source. These can range from behavior that is not software-specific, such as the inability to ignore sunk costs (i.e., wanting to modify an existing piece of code, they wrote earlier, rather than throw it away and starting again; although this does not seem to apply to throwing away code written by other people), to the use of the idioms of another language when writing in C.
- *Technically based.* Most existing education and training in software development tends to be based on purely technical issues. Economic issues are not usually raised formally, although informally time-to-completion is recognized as an important issue.

Unfortunately, once most developers have learned an initial set of attribute values and weightings for source code constructs, there is usually a long period of time before any subsequent major tuning or relearning takes place. Developers tend to be too busy applying their knowledge to question many of the underlying assumptions they have picked up along the way.

Based on this background, it is to be expected that many developers will harbor a few *myths* about what constitutes a good coding decision in certain circumstances. These coding guidelines cannot address all coding myths. Where appropriate, coding myths commonly encountered by your author are discussed.

14.5.5 Having to justify decisions

justifying decisions

Studies have found that having to justify a decision can affect the choice of decision-making strategy to be used. For instance, Tetlock and Boettger[1338] found that subjects who were accountable for their decisions used a much wider range of information in making judgments. While taking more information into account did not necessarily result in better decisions, it did mean that additional information that was both irrelevant and relevant to the decision was taken into account.

It has been proposed, by Tversky,[1372] that the elimination-by-aspects heuristic is easy to justify. However, while use of this heuristic may make for easier justification, it need not make for more accurate decisions.

A study performed by Simonson[1242] showed that subjects who had difficulty determining which alternative had the greatest utility tended to select the alternative that supported the best overall reasons (for choosing it).

Tetlock[1337] included an accountability factor into decision-making theory. One strategy that handles accountability as well as minimizing cognitive effort is to select the alternative that the perspective audience (i.e., code review members) is thought most likely to select. Not knowing which alternative they are likely to select can lead to a more flexible approach to strategies. The exception occurs when a person has already made the decision; in this case the cognitive effort goes into defending that decision.

During a code review, a developer may have to justify why a particular decision was made. While developers know that time limits will make it very unlikely that they will have to justify every decision, they do not know in advance which decisions will have to be justified. In effect, the developer will feel the need to be able to justify most decisions.

Requiring developers to justify why they have not followed a particular guideline recommendation can be a two-edged sword. Developers can respond by deciding to blindly follow guidelines (the path of least resistance), or they can invest effort in evaluating, and documenting, the different alternatives (not necessarily a good thing since the invested effort may not be warranted by the expected benefits). The extent to which some people will blindly obey authority was chillingly demonstrated in a number of studies by Milgram.[934]

14.6 Another theory about decision making

The theory that selection of a decision-making strategy is based on trading off cognitive effort and accuracy is not the only theory that has been proposed. Hammond, Hamm, Grassia, and Pearson[538] proposed that analytic decision making is only one end of a continuum; at the other end is intuition. They performed a study, using highway engineers, involving three tasks. Each task was designed to have specific characteristics (see Table 0.12). One task contained intuition-inducing characteristics, one analysis-inducing, and the third an equal mixture of the two. For the problems studied, intuitive cognition outperformed analytical cognition in terms of the empirical accuracy of the judgments.

Table 0.12: Inducement of intuitive cognition and analytic cognition, by task conditions. Adapted from Hammond.[538]

Task Characteristic	Intuition-Inducing State of Task Characteristic	Analysis-Inducing State of Task Characteristic
Number of cues	Large (>5)	Small
Measurement of cues	Perceptual measurement	Objective reliable measurement
Distribution of cue values	Continuous highly variable distribution	Unknown distribution; cues are dichotomous; values are discrete
Redundancy among cues	High redundancy	Low redundancy
Decomposition of task	Low	High
Degree of certainty in task	Low certainty	High certainty
Relation between cues and criterion	Linear	Nonlinear
Weighting of cues in environmental model	Equal	Unequal
Availability of organizing principle	Unavailable	Available
Display of cues	Simultaneous display	Sequential display
Time period	Brief	Long

One of the conclusions that Hammond et al. drew from these results is that "Experts should increase their awareness of the correspondence between task and cognition". A task having intuition-inducing characteristics is most likely to be out carried using intuition, and similarly for analysis-inducing characteristics.

Many developers sometimes talk of writing code intuitively. Discussion of intuition and flow of consciousness are often intermixed. The extent to which either intuitive or analytic decision making (if that is how developers operate) is more cost effective, or practical, is beyond this author's ability to even start to answer. It is mentioned in this book because there is a bona fide theory that uses these concepts and developers sometimes also refer to them.

0 developer flow

Intuition can be said to be characterized by rapid data processing, low cognitive control (the consistency with which a judgment policy is applied), and low awareness of processing. Its opposite, analysis, is characterized by slow data processing, high cognitive control, and high awareness of processing.

developer intuition

15 Expertise

People are referred to as being experts, in a particular domain, for several reasons, including:

expertise

- Well-established figures, perhaps holding a senior position with an organization heavily involved in that domain.
- Better at performing a task than the average person on the street.
- Better at performing a task than most other people who can also perform that task.
- Self-proclaimed experts, who are willing to accept money from clients who are not willing to take responsibility for proposing what needs to be done.[658]

Schneider[1202] defines a high-performance skill as one for which (1) more than 100 hours of training are required, (2) substantial numbers of individuals fail to develop proficiency, and (3) the performance of an expert is qualitatively different from that of the novice.

In this section, we are interested in why some people (the experts) are able to give a task performance that is measurably better than a non-expert (who can also perform the task).

There are domains in which those acknowledged as experts do not perform significantly better than those considered to be non-experts.[193] For instance, in typical cases the performance of medical experts was not much greater than those of doctors after their first year of residency, although much larger differences were seen for difficult cases. Are there domains where it is intrinsically not possible to become significantly better than one's peers, or are there other factors that can create a large performance difference between expert and non-expert performances? One way to help answer this question is to look at domains where the gap between expert and non-expert performance can be very large.

It is a commonly held belief that experts have some innate ability or capacity that enables them to do what they do so well. Research over the last two decades has shown that while innate ability can be a factor in performance (there do appear to be genetic factors associated with some athletic performances), the main factor in acquiring expert performance is time spent in *deliberate practice*.[393]

Deliberate practice is different from simply performing the task. It requires that people monitor their practice with full concentration and obtain feedback[582] on what they are doing (often from a professional teacher). It may also involve studying components of the skill in isolation, attempting to improve on particular aspects. The goal of this practice being to improve performance, not to produce a finished product.

Studies of the backgrounds of recognized experts, in many fields, found that the elapsed time between them starting out and carrying out their best work was at least 10 years, often with several hours of deliberate practice every day of the year. For instance, Ericsson, Krampe, and Tesch-Romer[394] found that, in a study of violinists (a perceptual-motor task), by age 20 those at the top level had practiced for 10,000 hours, those at the next level down 7,500 hours, and those at the lowest level of expertise had practiced for 5,000 hours. They also found similar quantities of practice being needed to attain expert performance levels in purely mental activities (e.g., chess).

People often learn a skill for some purpose (e.g., chess as a social activity, programming to get a job) without the aim of achieving expert performance. Once a certain level of proficiency is achieved, they stop trying to learn and concentrate on using what they have learned (in work, and sport, a distinction is made between training for and performing the activity). During everyday work, the goal is to produce a product or to provide a service. In these situations people need to use well-established methods, not try new (potentially dead-end, or leading to failure) ideas to be certain of success. Time spent on this kind of practice does not lead to any significant improvement in expertise, although people may become very fluent in performing their particular subset of skills.

What of individual aptitudes? In the cases studied by researchers, the effects of aptitude, if there are any, have been found to be completely overshadowed by differences in experience and deliberate practice times. What makes a person willing to spend many hours, every day, studying to achieve expert performance is open to debate. Does an initial aptitude or interest in a subject lead to praise from others (the path to musical and chess expert performance often starts in childhood), which creates the atmosphere for learning, or are other issues involved? IQ does correlate to performance during and immediately after training, but the correlation reduces over the years. The IQ of experts has been found to be higher than the average population at about the level of college students.

In many fields expertise is acquired by memorizing a huge amount of, domain-specific, knowledge and having the ability to solve problems using pattern-based retrieval on this knowledge base. The knowledge is structured in a form suitable for the kind of information retrieval needed for problems in a domain.[395]

A study by Carlson, Khoo, Yaure, and Schneider[200] examined changes in problem-solving activity as subjects acquired a skill (trouble shooting problems with a digital circuit). Subjects started knowing nothing, were given training in the task, and then given 347 problems to solve (in 28 individual, two-hour sessions, over a 15-week period). The results showed that subjects made rapid improvements in some areas (and little thereafter), extended practice produced continuing improvement in some of the task components, subjects acquired the ability to perform some secondary tasks in parallel, and transfer of skills to new digital circuits was substantial but less than perfect. Even after 56 hours of practice, the performance of subjects continued to show improvements and had not started to level off. Where are the limits to continued improvements? A study by Crossman[300] of workers producing cigars showed performance improving according to the power law of practice for the first five years of employment. Thereafter performance improvements slow; factors cited for this slow down include approaching the speed limit of the equipment being used and the capability of the musculature of the workers.

power law of learning 0

15.1 Knowledge

developer knowledge

A distinction is often made between different kinds of knowledge. Declarative knowledge are the facts; procedural knowledge are the skills (the ability to perform learned actions). Implicit memory is defined as

memory without conscious awareness— it might be considered a kind of knowledge.

0 implicit learning

15.1.1 Declarative knowledge

This consists of knowledge about facts and events. For instance, the keywords used to denote the integer types are **char**, **short**, **int**, and **long**. This kind of knowledge is usually explicit (we know what we know), but there are situations where it can be implicit (we make use of knowledge that we are not aware of having[849]). The coding guideline recommendations in this book have the form of declarative knowledge.

declarative knowledge

It is the connections and relationships between the individual facts, for instance the relative sizes of the integer types, that differentiate experts from novices (who might know the same facts). This kind of knowledge is rather like web pages on the Internet; the links between different pages corresponding to the connections between facts made by experts. Learning a subject is more about organizing information and creating connections between different items than it is about remembering information in a rote-like fashion.

This was demonstrated in a study by McKeithen, Reitman, Ruster, and Hirtle,[917] who showed that developers with greater experience with a language organized their knowledge of language keywords in a more structured fashion. Education can provide the list of facts, it is experience that provides the connections between them.

The term *knowledge base* is sometimes used to describe a collection of information and links about a given topic. The C Standard document is a knowledge base. Your author has a C knowledge base in his head, as do you the reader. This book is another knowledge base dealing with C. The difference between this book and the C Standard document is that it contains significantly more explicit links connecting items, and it also contains information on how the language is commonly implemented and used.

15.1.2 Procedural knowledge

This consists of knowledge about how to perform a task; it is often implicit.

procedural knowledge

Knowledge can start off by being purely declarative and, through extensive practice, becomes procedural; for instance, the process of learning to drive a car. An experiment by Sweller, Mawer, and Ward[1325] showed how subjects' behavior during mathematical problem solving changed as they became more proficient. This suggested that some aspects of what they were doing had been proceduralized.

Some of the aspects of writing source code that can become proceduralized are discussed elsewhere.

0 developer flow
0 automatization

15.2 Education

What effect does education have on people who go on to become software developers?

developer education

> *Education should not be thought of as replacing the rules that people use for understanding the world but rather as introducing new rules that enter into competition with the old ones. People reliably distort the new rules in the direction of the old ones, or ignore them altogether except in the highly specific domains in which they were taught.*

Page 206 of Holland et al.[585]

Education can be thought of as trying to do two things (of interest to us here)— teach students skills (procedural knowledge) and providing them with information, considered important in the relevant field, to memorize (declarative knowledge). To what extent does education in subjects not related to software development affect a developer's ability to write software?

Some subjects that are taught to students are claimed to teach general reasoning skills; for instance, philosophy and logic. There are also subjects that require students to use specific reasoning skills, for instance statistics requires students to think probabilistically. Does attending courses on these subjects actually have any measurable effect on students' capabilities, other than being able to answer questions in an exam. That is, having acquired some skill in using a particular system of reasoning, do students apply it outside of the domain in which they learnt it? Existing studies have supplied a *No* answer to this question.[922, 1011] This *No* was even found to apply to specific skills; for instance, statistics (unless the problem explicitly involves statistical thinking within the applicable domain) and logic.[224]

A study by Lehman, Lempert, and Nisbett[831] measured changes in students' statistical, methodological, and conditional reasoning abilities (about everyday-life events) between their first and third years. They

found that both psychology and medical training produced large effects on statistical and methodological reasoning, while psychology, medical, and law training produced effects on the ability to perform conditional reasoning. Training in chemistry had no affect on the types of reasoning studied. An examination of the skills taught to students studying in these fields showed that they correlated with improvements in the specific types of reasoning abilities measured. The extent to which these reasoning skills transferred to situations that were not everyday-life events was not measured. Many studies have found that in general people do not transfer what they have learned from one domain to another.

expertise 0
transfer to another domain

It might be said that passing through the various stages of the education process is more like a filter than a learning exercise. Those that already have the abilities being the ones that succeed.[1401] A well-argued call to arms to improve students' general reasoning skills, through education, is provided by van Gelder.[1400]

Good education aims to provide students with an overview of a subject, listing the principles and major issues involved; there may be specific cases covered by way of examples. Software development does require knowledge of general principles, but most of the work involves a lot of specific details: specific to the application, the language used, and any existing source code, while developers may have been introduced to the C language as part of their education. The amount of exposure is unlikely to have been sufficient for the building of any significant knowledge base about the language.

15.2.1 Learned skills

Education provides students with *learned knowledge*, which relates to the title of this subsection *learned skills*. Learning a skill takes practice. Time spent by students during formal education practicing their programming skills is likely to total less than 60 hours. Six months into their first development job they could very well have more than 600 hours of experience. Although students are unlikely to complete their education with a lot of programming experience, they are likely to continue using the programming beliefs and practices they have acquired. It is not the intent of this book to decry the general lack of good software development training, but simply to point out that many developers have not had the opportunity to acquire good habits, making the use of coding guidelines even more essential.

developer 0
expertise

Can students be taught in a way that improves their general reasoning skills? This question is not directly relevant to the subject of this book; but given the previous discussion, it is one that many developers will be asking. Based on the limited researched carried out to date the answer seems to be yes. Learning requires intense, quality practice. This would be very expensive to provide using human teachers, and researchers are looking at automating some of the process. Several automated training aids have been produced to help improve students' reasoning ability and some seem to have a measurable affect.[1401]

15.2.2 Cultural skills

Cultural skills include the use of language and category formation. Nisbett and Norenzayan[1015] provide an overview of culture and cognition. A more practical guide to cultural differences and communicating with people from different cultures, from the perspective of US culture, is provided by Wise, Hannaman, Kozumplik, Franke, and Leaver.[1475]

developer 792
language and culture
categorization 0
cultural differences

15.3 Creating experts

To become an expert a person needs motivation, time, economic resources, an established body of knowledge to learn from, and teachers to guide.

One motivation is to be the best, as in chess and violin playing. This creates the need to practice as much as others at that level. Ericsson found[394] that four hours per day was the maximum concentrated training that people could sustain without leading to exhaustion and burnout. If this is the level of commitment, over a 10-year period, that those at the top have undertaken, then anybody wishing to become their equal will have to be equally committed. The quantity of practice needed to equal expert performance in less competitive fields may be less. One should ask of an expert whether they attained that title because they are simply as good as the best, or because their performance is significantly better than non-experts.

In many domains people start young, between three and eight in some cases,[394] their parents' interest being critical in providing equipment, transport to practice sessions, and the general environment in which to

study.

An established body of knowledge to learn from requires that the domain itself be in existence and relatively stable for a long period of time. The availability of teachers requires a domain that has existed long enough for them to have come up through the ranks; and one where there are sufficient people interested in it that it is possible to make as least as much from teaching as from performing the task.

The research found that domains in which the performance of experts was not significantly greater than non-experts lacked one or more of these characteristics.

15.3.1 Transfer of expertise to different domains

expertise transfer to another domain

Research has shown that expertise within one domain does not confer any additional skills within another domain.[33] This finding has been duplicated for experts in real-world domains, such as chess, and in laboratory-created situations. In one series of experiments, subjects who had practiced the learning of sequences of digits (after 50–100 hours of practice they could commit to memory, and recall later, sequences containing more than 20 digits) could not transfer their expertise to learning sequences of other items.[217]

15.4 Expertise as mental set

Software development is a new field that is still evolving at a rapid rate. Most of the fields in which expert performance has been studied are much older, with accepted bodies of knowledge, established traditions, and methods of teaching.

Sometimes knowledge associated with software development does not change wholesale. There can be small changes within a given domain; for instance, the move from K&R C to ISO C.

In a series of experiments Wiley,[1465] showed that in some cases non-experts could outperform experts within their domain. She showed that an expert's domain knowledge can act as a mental set that limits the search for a solution; the expert becomes fixated within the domain. Also, in cases where a new task does not fit the pattern of highly proceduralized behaviors of an expert, a novice's performance may be higher.

15.5 Software development expertise

software development expertise

Given the observation that in some domains the acknowledged experts do not perform significantly better than non-experts, we need to ask if it is possible that any significant performance difference could exist in software development. Stewart and Lusk[1297] proposed a model of performance that involves seven components. The following discussion breaks down expertise in software development into five major areas.

1. *Knowledge (declarative) of application domain.* Although there are acknowledged experts in a wide variety of established application domains, there are also domains that are new and still evolving rapidly. The use to which application expertise, if it exists, can be put varies from high-level design to low-level algorithmic issues (i.e., knowing that certain cases are rare in practice when tuning a time-critical section of code).

2. *Knowledge (declarative) of algorithms and general coding techniques.* There exists a large body of well-established, easily accessible, published literature about algorithms. While some books dealing with general coding techniques have been published, they are usually limited to specific languages, application domains (e.g., embedded systems), and often particular language implementations. An important issue is the rigor with which some of the coding techniques have been verified; it often leaves a lot to be desired, including the level of expertise of the author.

3. *Knowledge (declarative) of programming language.* The C programming language is regarded as an established language. Whether 25 years is sufficient for a programming language to achieve the status of being established, as measured by other domains, is an open question. There is a definitive document, the ISO Standard, that specifies the language. However, the sales volume of this document has been extremely low, and most of the authors of books claiming to teach C do not appear to have read the standard. Given this background, we cannot expect any established community of expertise in the C language to be very large.

4. *Ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms.* Procedural knowledge is acquired through practice. While university students may have had access to computers since the 1970s, access for younger people did not start to occur until the mid 1980s. It is possible for developers to have had 25 years of software development practice.
5. *Development environment.* The development environment in which people have to work is constantly changing. New versions of operating systems are being introduced every few years; new technologies are being created and old ones are made obsolete. The need to keep up with development is a drain on resources, both in intellectual effort and in time. An environment in which there is a rapid turnover in applicable knowledge and skills counts against the creation of expertise.

Although the information and equipment needed to achieve a high-level of expertise might be available, there are several components missing. The motivation to become the best software developer may exist in some individuals, but there is no recognized measure of what *best* means. Without the measuring and scoring of performances it is not possible for people to monitor their progress, or for their efforts to be rewarded. While there is a demand for teachers, it is possible for those with even a modicum of ability to make substantial amounts of money doing (not teaching) development work. The incentives for good teachers are very poor.

Given this situation we would not expect to find large performance differences in software developers through training. If training is insufficient to significantly differentiate developers the only other factor is individual ability. It is certainly your author's experience— individual ability is a significant factor in a developer's performance.

Until the rate of change in general software development slows down, and the demand for developers falls below the number of competent people available, it is likely that ability will continue to the dominant factor (over training) in developer performance.

15.6 Software developer expertise

developer expertise

Having looked at expertise in general and the potential of the software development domain to have experts, we need to ask how expertise might be measured in people who develop software. Unfortunately, there are no reliable methods for measuring software development expertise currently available. However, based on the previously discussed issues, we can isolate the following technical competencies (social competencies[1007] are not covered here, although they are among the skills sought by employers,[83] and software developers have their own opinions[837, 1268]):

- Knowledge (declarative) of application domain.
- Knowledge (declarative) of algorithms and general coding techniques.
- Knowledge (declarative) of programming languages.
- Cognitive ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms (a specialized form of general analytical and conceptual thinking).
- Knowledge (metacognitive) about knowledge (i.e., judging the quality and quantity of one's expertise).

Your author has firsthand experience of people with expertise individually within each of these components, while being non-experts in all of the others. People with application-domain expertise and little programming knowledge or skill are relatively common. Your author once implemented the semantics phase of a CHILL (Communications HIgh Level Language) compiler and acquired expert knowledge in the semantics of that language. One day he was shocked to find he could not write a CHILL program without reference to some existing source code (to *refresh* his memory of general program syntax); he had acquired an extensive knowledge based of the semantics of the language, but did not have the procedural knowledge needed to write a program (the compiler was written in another language).[0.6]

[0.6] As a compiler writer, your author is sometimes asked to help fix problems in programs written in languages he has never seen before (how can one be so expert and not know every language?). He now claims to be an expert at comprehending programs written in unknown languages for application domains he knows nothing about (he is helped by the fact that few languages have any truly unique constructs).

A developer's knowledge of an application domain can only be measured using the norms of that domain. One major problem associated with measuring overall developer expertise is caused by the fact that different developers are likely to be working within different domains. This makes it difficult to cross correlate measurements.

A study at Bell Labs[330] showed that developers who had worked on previous releases of a project were much more productive than developers new to a project. They divided time spent by developers into discovery time (finding out information) and work time (doing useful work). New project members spent 60% to 80% of their time in discovery and 20% to 40% doing useful work. Developers experienced with the application spent 20% of their time in discovery and 80% doing useful work. The results showed a dramatic increase in efficiency (useful work divided by total effort) from having been involved in one project cycle and less dramatic an increase from having been involved in more than one release cycle. The study did not attempt to separate out the kinds of information being sought during discovery.

Another study at Bell Labs[952] found that the probability of a fault being introduced into an application, during an update, correlated with the experience of the developer doing the work. More experienced developers seemed to have acquired some form of expertise in an application that meant they were less likely to introduce a fault into it.

A study of development and maintenance costs of programs written in C and Ada[1504] found no correlation between salary grade (or employee rating) and rate of bug fix/add feature rate.

Your author's experience is that developers' general knowledge of algorithms (in terms of knowing those published in well-known text-books) is poor. There is still a strongly held view, by developers, that it is permissible for them to invent their own ways of doing things. This issue is only of immediate concern to these coding guidelines as part of the widely held, developers', belief that they should be given a free hand to write source as they see fit.

There is a group of people who might be expected to be experts in a particular programming languages—those who have written a compiler for it (or to be exact those who implemented the semantics phase of the compiler, anybody working on others parts [e.g., code generation] does not need to acquire detailed knowledge of the language semantics). Your author knows a few people who are C language experts and have not written a compiler for that language. Based on your author's experience of implementing several compilers, the amount of study needed to be rated as an expert in one computer language is approximately 3 to 4 hours per day (not even compiler writers get to study the language for every hour of the working day; there are always other things that need to be attended to) for a year. During that period, every sentence in the language specification will be read and analyzed in detail several times, often in discussion with colleagues. Generally developer knowledge of the language they write in is limited to the subset they learned during initial training, perhaps with some additional constructs learned while reading other developers' source or talking to other members of a project. The behavior of the particular compiler they use also colors their view of those constructs.

Expertise in the act of comprehending and writing software is hard to separate from knowledge of the application domain. There is rarely any need to understand a program without reference to the application domain it was written for. When computers were centrally controlled, before the arrival of desktop computers, many organizations offered a programming support group. These support groups were places where customers of the central computer (usually employees of the company or staff at a university) could take programs they were experiencing problems with. The staff of such support groups were presented with a range of different programs for which they usually had little application-domain knowledge. This environment was ideal for developing program comprehension skills without the need for application knowledge (your author used to take pride in knowing as little as possible about the application while debugging the presented programs). Such support groups have now been reduced to helping customers solve problems with packaged software. Environments in which pure program-understanding skills can be learned now seem to have vanished.

What developers do is discussed elsewhere. An expert developer could be defined as a person who is 0 developers what do they do? able to perform these tasks better than the majority of their peers. Such a definition is open-ended (how is

productivity 0
developer

better defined for these tasks?) and difficult to measure. In practice, it is productivity that is the sought-after attribute in developers.

Some studies have looked at how developers differ (which need not be the same as measuring expertise), including their:

developers 0
organized knowledge
developer 0
personality

- ability to remember more about source code they have seen,
- personality differences,
- knowledge of the computer language used, and
- ability to estimate the effort needed to implement the specified functionality.[693]

A study by Jørgensen and Sjøberg[694] looked at maintenance tasks (median effort 16-work hours). They found that developers' skill in predicting maintenance problems improved during their first two years on the job; thereafter there was no correlation between increased experience (average of 7.7 years' development experience, 3.4 years on maintenance of the application) and increased skill. They attributed this lack of improvement in skill to a lack of learning opportunities (in the sense of deliberate practice and feedback on the quality of their work).

Job advertisements often specify that a minimum number of years of experience is required. Number of years is known not to be a measure of expertise, but it provides some degree of comfort that a person has had to deal with many of the problems that might occur within a given domain.

15.6.1 Is software expertise worth acquiring?

Most developers are not professional programmers any more than they are professional typists. Reading and writing software is one aspect of their job. The various demands on their time is such that most spend a small portion of their time writing software. Developers need to balance the cost of spending time becoming more skillful programmers against the benefits of possessing that skill. Experience has shown that software can be written by relatively unskilled developers. One consequence of this is that few developers ever become experts in any computer language.

When estimating benefit over a relatively short period of time, time spent learning more about the application domain frequently has a greater return than honing programming skills.

15.7 Coding style

coding guidelines
coding style

As an Englishman, your author can listen to somebody talking and tell if they are French, German, Australian, or one of many other nationalities (and sometimes what part of England they were brought up in). From what they say, I might make an educated guess about their educational level. From their use of words like *cool*, *groovy*, and so on, I might guess age and past influences (young or ageing hippie).

source code
accent

Source code written by an experienced developer sometimes has a recognizable style. Your author can often tell if a developer's previous language was Fortran, Pascal, or Basic. But he cannot tell if their previous language was Lisp or APL (anymore than he can distinguish regional US accents, nor can many US citizens tell the difference among an English, Scottish, Irish, or Australian accent), because he has not had enough exposure to those languages.

Is coding style a form of expertise (a coherent framework that developers use to express their thoughts), or is it a ragbag of habits that developers happen to have? Programs have been written that can accurately determine the authorship of C source code (success rates of 73% have been reported[775]). These experiments used, in part, source code written by people new to software development (i.e., students). Later work using neural networks[776] was able to get the failure rate down to 2%. That it was possible to distinguish programs written by very inexperienced developers suggests that style might simply be a ragbag of habits (these developers not having had time to put together a coherent way of writing source).

The styles used by inexperienced developers can even be detected after an attempt has been made to hide the original authorship of the source. Plagiarism is a persistent problem in many universities' programming courses and several tools have been produced that automatically detect source code plagiarisms.[1120, 1417]

One way for a developer to show mastery of coding styles would be to have the ability to write source using a variety of different styles, perhaps even imitating the style of others. The existing author analysis tools are being used to verify that different, recognizable styles were being used.

It was once thought (and still is by some people) that there is a correct way to speak. Received Pronunciation (as spoken on the BBC many years ago) was once promoted as correct usage within the UK.

Similarly, many people believe that source code can be written in a good style or a bad style. A considerable amount of time has been, and will probably continue to be, spent discussing this issue. Your authors' position is the following:

- Identifiable source code styles exist.
- It is possible for people to learn new coding styles.
- It is very difficult to explain style to non-expert developers.
- Learning a new style is sufficiently time-consuming, and the benefits are likely to be sufficiently small, that a developer is best advised to invest effort elsewhere.

Students of English literature learn how to recognize writing styles. There are many more important issues that developers need to learn before they reach the stage where learning about stylistic issues becomes worthwhile.

The phrase coding guidelines and coding style are sometimes thought of, by developers of as being synonymous. This unfortunate situation has led to coding guidelines acquiring a poor reputation. While recognizing the coding style does exist, they are not the subject of these coding guidelines. The term *existing practice* refers to the kinds of constructs often found in existing programs. Existing practice is dealt with as an issue in its own right, independent of any concept of style.

0 coding guidelines introduction

16 Human characteristics

human characteristics

Humans are not ideal machines, an assertion that may sound obvious. However, while imperfections in physical characteristics are accepted, any suggestion that the mind does not operate according to the laws of mathematical logic is rarely treated in the same forgiving way. For instance, optical illusions are accepted as curious anomalies of the eye/brain system; there is no rush to conclude that human eyesight is faulty.

Optical illusions are often the result of preferential biases in the processing of visual inputs that, in most cases, are beneficial (in that they simplify the processing of ecologically common inputs). In Figure 0.19, which of the two squares indicated by the arrows is the brighter one? Readers can verify that the indicated squares have exactly the same grayscale level. Use a piece of paper containing two holes, that display only the two squares pointed to.

This effect is not caused by low-level processing, by the brain, of the input from the optic nerve; it is caused by high-level processing of the scene (recognizing the recurring pattern and that some squares are within a shadow). Anomalies caused by this high-level processing are not limited to grayscales. The brain is thought to have specific areas dedicated to the processing of faces. The, so-called, *Thatcher illusion* is an example of this special processing of faces. The two faces in Figure 0.20 look very different; turn the page upside down and they look almost identical.

Music is another input stimulus that depends on specific sensory input/brain affects occurring. There is no claim that humans cannot hear properly, or that they should listen to music derived purely from mathematical principles.

Studies have uncovered situations where the behavior of human cognitive processes does not correspond to some generally accepted norm, such as Bayesian inference. However, it cannot be assumed that cognitive limitations are an adaption to handle the physical limitations of the brain. There is evidence to suggest that some of these so-called cognitive limitations provide near optimal solutions for some real-world problems.[570]

0 evolutionary psychology

The ability to read, write, and perform complex mathematical reasoning are very recent (compared to several million years of evolution) cognitive skill requirements. Furthermore, there is no evidence to suggest

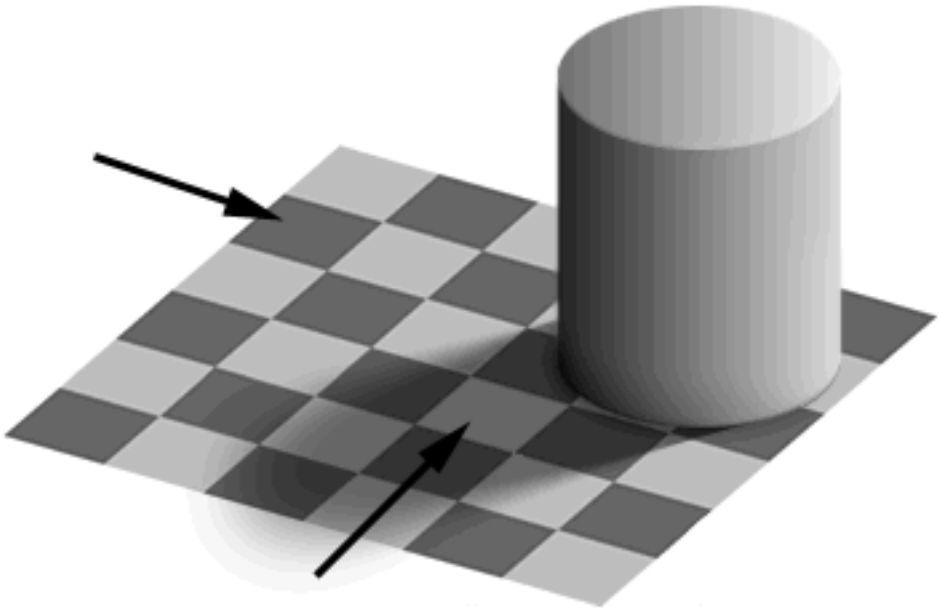


Figure 0.19: Checker shadow (by Edward Adelson). Which of the two squares indicated by the arrows is the brighter one (following inverted text gives answer)? Both squares reflect the same amount of light (this can be verified by covering all of squares except the two indicated), but the human visual system assigns a relative brightness that is consistent with the checker pattern.

Figure 0.20: The Thatcher illusion. With permission from Thompson.[1351] The facial images look very similar when viewed in one orientation and very different when viewed in another (turn page upside down).

that possessing these skills improves the chances of a person passing on their genes to subsequent generations (in fact one recent trend suggests otherwise[1236]). So we should not expect human cognitive processes to be tuned for performing these activities.

Table 0.13: Cognitive anomalies. Adapted from McFadden.[914]

Effect	Description
CONTEXT	
Anchoring	Judgments are influenced by quantitative cues contained in the statement of the decision task
Context	Prior choices and available options in the decision task influence perception and motivation
Framing	Selection between mathematically equivalent solutions to a problem depends on how their outcome is framed.
Prominence	The format in which a decision task is stated influences the weight given to different aspects
REFERENCE POINT	
Risk asymmetry	Subjects show risk-aversion for gains, risk-preference for losses, and weigh losses more heavily
Reference point	Choices are evaluated in terms of changes from an endowment or status quo point
Endowment	Possessed goods are valued more highly than those not possessed; once a function has been written
developers are loath to throw it away and start again	
AVAILABILITY	
Availability	Responses rely too heavily on readily retrievable information and too little on background information
Certainty	Sure outcomes are given more weight than uncertain outcomes
Experience	Personal history is favored relative to alternatives not experienced
Focal	Quantitative information is retrieved or reported categorically
Isolation	The elements of a multiple-part or multi-stage lottery are evaluated separately
Primacy and Recency	Initial and recently experienced events are the most easily recalled
Regression	Idiosyncratic causes are attached to past fluctuations, and regression to the mean is underestimated
Representativeness	High conditional probabilities induce overestimates of unconditional probabilities
Segregation	Lotteries are decomposed into a sure outcome and a gamble relative to this sure outcome
SUPERSTITION	
Credulity	Evidence that supports patterns and causal explanations for coincidences is accepted too readily
Disjunctive	Consumers fail to reason through or accept the logical consequences of actions
Superstition	Causal structures are attached to coincidences, and "quasi-magical" powers to opponents
Suspicion	Consumers mistrust offers and question the motives of opponents, particularly in unfamiliar situations
PROCESS	
Rule-Driven	Behavior is guided by principles, analogies, and exemplars rather than utilitarian calculus
Process	Evaluation of outcomes is sensitive to process and change
Temporal	Time discounting is temporally inconsistent, with short delays discounted too sharply relative to long delays
PROJECTION	
Misrepresentation	Subjects may misrepresent judgments for real or perceived strategic advantage
Projection	Judgments are altered to reinforce internally or project to others a self-image

Table 0.13 lists some of the cognitive anomalies (difference between human behavior and some idealized norm) applicable to writing software. There are other cognitive anomalies, some of which may also be applicable, and others that have limited applicability; for instance, writing software is a private, not a social activity. Cognitive anomalies relating to herd behavior and conformity to social norms are unlikely to be of

interest.

16.1 Physical characteristics

developer physical characteristics

Before moving on to the main theme of this discussion, something needs to be said about physical characteristics.

The brain is the processor that the software of the mind executes on. Just as silicon-based processors have special units that software can make use of (e.g., floating point), the brain appears to have special areas that perform specific functions.[1088] This book treats the workings of the brain/mind combination as a black box. We are only interested in the outputs, not the inner workings (brain-imaging technology has not yet reached the stage where we can deduce functionality by watching the signals travelling along neurons).

Eyes are the primary information-gathering sensors for reading and writing software. A lot of research has been undertaken on how the eyes operate and interface with the brain.[1047] Use of other information-gathering sensors has been proposed, hearing being the most common (both spoken and musical[1421]). These are rarely used in practice, and they are not discussed further in this book.

typing 792 mistakes

Hands/fingers are the primary output-generation mechanism. A lot of research on the operation of limbs has been undertaken. The impact of typing on error rate is discussed elsewhere.

Developers are assumed to be physically mature (we do not deal with code written by children or adolescents) and not to have any physical (e.g., the impact of dyslexia on reading source code is not known; another unknown is the impact of deafness on a developer's ability to abbreviate identifiers based on their sound) or psychiatric problems.

Issues such as genetic differences (e.g., male vs. female[1111]) or physical changes in the brain caused by repeated use of some functional unit (e.g., changes in the hippocampi of taxi drivers[887]) are not considered here.

16.2 Mental characteristics

developer mental characteristics

memory 0 developer

This section provides an overview of those mental characteristics that might be considered important in reading and writing software. Memory, particularly short-term memory, is an essential ability. It might almost be covered under physical characteristics, but knowledge of its workings has not quite yet reached that level of understanding. An overview of the characteristics of memory is given in the following subsection. The consequences of these characteristics are discussed throughout the book.

The idealization of developers aspiring to be omnipotent logicians gets in the way of realistically approaching the subject of how best to make use of the abilities of the human mind. Completely rational, logical, and calculating thought may be considered to be the ideal tools for software development, but they are not what people have available in their heads. Builders of bridges do not bemoan the lack of unbreakable materials available to them, they have learned how to work within the limitations of the materials available. This same approach is taken in this book, work with what is available.

This overview is intended to provide background rationale for the selection of, some, coding guidelines. In some cases, this results in recommendations against the use of constructs that people are likely to have problems processing correctly. In other cases this results in recommendations to do things in a particular way. These recommendations could be based on, for instance, capacity limitations, biases/heuristics (depending on the point of view), or some other cognitive factors.

Some commentators recommend that ideal developer characteristics should be promoted (such ideals are often accompanied by a list of tips suggesting activities to perform to help achieve these characteristics, rather like pumping iron to build muscle). This book contains no exhortations to try harder, or tips on how to become better developers through mental exercises. In this book developers are taken as they are, not some idealized vision of how they should be.

Hopefully the reader will recognize some of the characteristics described here in themselves. The way forward is to learn to deal with these characteristics, not to try to change what could turn out to be intrinsic properties of the human brain/mind.

Software development is not the only profession for which the perceived attributes of practitioners do not

correspond to reality. Darley and Batson[317] performed a study in which they asked subjects (theological seminary students) to walk across campus to deliver a sermon. Some of the subjects were told that they were late and the audience was waiting, the remainder were not told this. Their journey took them past a *victim* moaning for help in a doorway. Only 10% of subjects who thought they were late stopped to help the victim; of the other subjects 63% stopped to help. These results do not match the generally perceived behavior pattern of theological seminary students.

Most organizations do not attempt to measure mental characteristics in developer job applicants; unlike many other jobs for which individual performance can be an important consideration. Whether this is because of an existing culture of not measuring, lack of reliable measuring procedures, or fear of frightening off prospective employees is not known.

16.2.1 Computational power of the brain

developer computational power

One commonly used method of measuring the performance of silicon-based processors is to quote the number of instructions (measured in millions) they can execute in a second. This is known to be an inaccurate measure, but it provides an estimate.

The brain might simply be a very large neural net, so there will be no instructions to count as such. Merkle[926] used various approaches to estimate the number of synaptic operations per second; the followings figures are taken from his article:

- Multiplying the number of synapses (10^{15}) by their speed of operation (about 10 impulses/second) gives 10^{16} synapse operations per second.
- The retina of the eye performs an estimated 10^{10} analog add operations per second. The brain contains 10^2 to 10^4 times as many nerve cells as the retina, suggesting that it can perform 10^{12} to 10^{14} operations per second.
- A total brain power dissipation of 25 watts (an estimated 10 watts of useful work) and an estimated energy consumption of 5×10^{-15} joules for the switching of a nerve cell membrane provides an upper limit of 2×10^{15} operations per second.

A synapse switching on and off is rather like a transistor switching on and off. They both need to be connected to other switches to create a larger functional unit. It is not known how many synapses are used to create functional units in the brain, or even what those functional units might be. The distance between synapses is approximately 1 mm. Simply sending a signal from one part of the brain to another part requires many synaptic operations, for instance, to travel from the front to the rear of the brain requires at least 100 synaptic operations to propagate the signal. So the number of synaptic operations per high-level, functional operation is likely to be high. Silicon-based processors can contain millions of transistors. The potential number of transistor-switching operations per second might be greater than 10^{14}, but the number of instructions executed is significantly smaller.

Although there have been studies of the information-processing capacity of the brain (e.g., visual attention,[1419] storage rate into long-term memory,[801] and correlations between biological factors and intelligence[1405]), we are a long way from being able to deduce the likely work rates of the components of the brain used during code comprehension. The issue of overloading the computational resources of the brain is discussed elsewhere.

0 cognitive effort

There are several executable models of how various aspects of human cognitive processes operate. The ACT-R model[35] has been applied to a wide range of problems, including learning, the visual interface, perception and action, cognitive arithmetic, and various deduction tasks.

Developers are familiar with the idea that a more powerful processor is likely to execute a program more quickly than a less powerful one. Experience shows that some minds are quicker at solving some problems than other minds and other problems (a correlation between what is known as *inspection time* and IQ has been found[335]). For these coding guidelines, speed of mental processing is not a problem in itself. The problem of limited processing resources operating in a time-constrained environment, leading to errors being

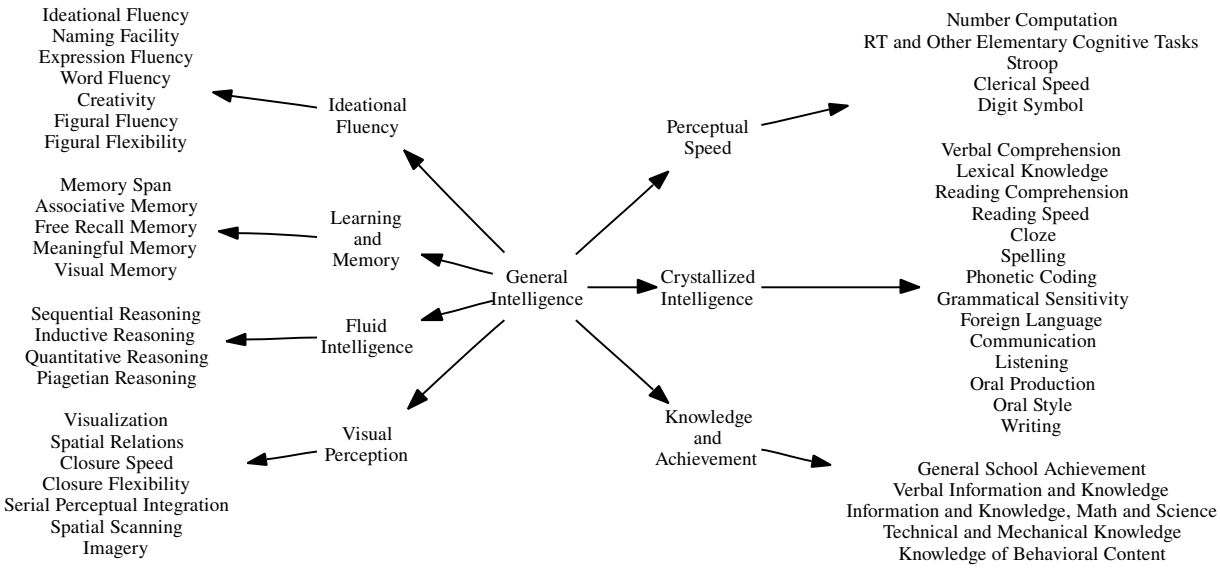


Figure 0.21: A list of and structure of ability constructs. Adapted from Ackerman.[2]

made, could be handled if the errors were easily predicted. It is the fact that different developers have ranges of different abilities that cause the practical problems. Developer A can have trouble understanding the kinds of problems another developer, B, could have understanding the code he, A, has written. The problem is how does a person, who finds a particular task easy, relate to the problems a person, who finds that task difficult, will have?

The term *intelligence* is often associated with performance ability (to carry out some action in a given amount of time). There has been a great deal of debate about what intelligence is, and how it can be measured. Gardner[471] argues for the existence of at least six kinds of intelligence— bodily kinesthetic, linguistic, mathematical, musical, personal, and spatial. Studies have shown that there can be dramatic differences between subjects rated high and low in these intelligences (linguistic[501] and spatial[883]). Ackerman and Heggestad[2] review the evidence for overlapping traits between intelligence, personality, and interests (see Figure 0.21). An extensive series of tests carried out by Süß, Oberauer, Wittmann, Wilhelm, and Schulze[1317] found that intelligence was highly correlated to working memory capacity. The strongest relationship was found for reasoning ability.

The failure of so-called intelligence tests to predict students' job success on leaving college or university is argued with devastating effect by McClelland,[909] who makes the point that the best testing is criterion sampling (for developers this would involve testing those attributes that distinguish *betterness* in developers). Until employers start to measure those employees who are involved in software development, and a theory explaining how these relate to the problem of developing software-based applications is available, there is little that can be said. At our current level of knowledge we can only say that developers having different abilities may exhibit different failure modes when solving problems.

16.2.2 Memory

memory developer

Studies have found that human memory might be divided into at least two (partially connected) systems, commonly known as *short-term memory* (STM) and *long-term memory* (LTM). The extent to which STM and LTM really are different memory systems, and not simply two ends of a continuum of memory properties, continues to be researched and debated. Short-term memory tends to operate in terms of speech sounds and have a very limited capacity; while long-term memory tends to be semantic- or episodic-based and is often treated as having an infinite capacity (a lifetime of memories is estimated to be represented in 10^9 bits;[801]

memory 0 episodic

this figure takes forgetting into account).

There are two kinds of query that are made against the contents of memory. During *recall* a person attempts to use information immediately available to them to access other information held in memory. During *recognition*, a person decides whether they have an existing memory for information that is being presented.

Much of the following discussion involves human memory performance with unchanging information. Developers often have to deal with changing information (e.g., the source code may be changing on a daily basis; the value of variables may be changing as developers run through the execution of code in their heads). Human memory performance has some characteristics that are specific to dealing with changing information.[295, 712] However, due to a lack of time and space, this aspect of developer memory performance is not covered in any detail in this book.

Miller 7±2

As its name implies, STM is an area of memory that stores information for short periods of time. For more than 100 years researchers have been investigating the properties of STM. Early researchers started by trying to measure its capacity. A paper by Miller[935] entitled *The magical number seven, plus or minus two: Some limits on our capacity for processing information* introduced the now-famous 7±2 rule. Things have moved on, during the 47 years since the publication of his paper[684] (not that Miller ever proposed 7±2 as the capacity of STM; he simply drew attention to the fact that this range of values fit the results of several experiments).

memory digit span

Readers might like to try measuring their STM capacity. Any Chinese-speaking readers can try this exercise twice, using the English and Chinese words for the digits.[591] Use of Chinese should enable readers to apparently increase the capacity of STM (explanation follows). The digits in the outside margin can be used. Slowly and steadily read the digits in a row, out loud. At the end of each row, close your eyes and try to repeat the sequence of digits in the same order. If you make a mistake, go on to the next row. The point at which you cannot correctly remember the digits in any two rows of a given length indicates your capacity limit— the number of digits in the previous rows.

8704
2193
3172
57301
02943
73619
659420
402586
542173
6849173
7931684
3617458
27631508
81042963
07239861
578149306
293486701
721540683
5762083941
4093067215
9261835740

Measuring working memory capacity using sequences of digits relies on several assumptions. It assumes that working memory treats all items the same way (what if letters of the alphabet had been used instead), and it also assumes that individual concepts are the unit of storage. Studies have shown that both these assumptions are incorrect. What the preceding exercise measured was the amount of *sound* you could keep in working memory. The sound used to represent digits in Chinese is shorter than in English. The use of Chinese should enable readers to maintain information on more digits (average 9.9[592]) using the same amount of sound storage. A reader using a language for which the sound of the digits is longer would be able to maintain information on fewer digits (e.g., average 5.8 in Welsh[384]). The average for English is 6.6.

Studies have shown that performance on the *digit span* task is not a good predictor of performance on other short- or long-term memory for items. However, a study by Martin[898] found that it did correlate with memory for the temporal occurrence of events.

In the 1970s Baddeley asked what purpose short-term memory served. He reasoned that its purpose was to act as a temporary area for activities such as mental arithmetic, reasoning, and problem solving. The model of *working memory* he proposed is shown in Figure 0.22. There are three components, each with its own independent temporary storage areas, each holding and using information in different ways.

What does the central executive do? It is assumed to be the system that handles attention, controlling the phonological loop, the visuo-spatial sketch pad, and the interface to long-term memory. The central executive needs to remember information while performing tasks such as text comprehension and problem solving. The potential role of this central executive is discussed elsewhere.

Sequences of single digits containing 4 to 10 digits.

Visual information held in the visuo-spatial sketch pad decays very rapidly. Experiments have shown that people can recall four or five items immediately after they are presented with visual information, but that this recall rate drops very quickly after a few seconds. From the source code reading point of view, the visuo-spatial sketch pad is only operative for the source code currently being looked at.

0 attention
visuo-spatial memory

While remembering digit sequences, readers may have noticed that the sounds used for them went around

phonological loop

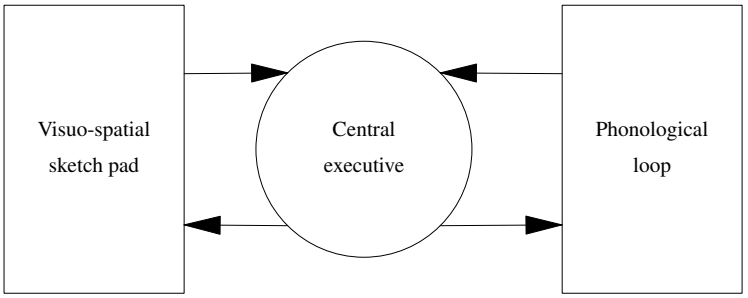


Figure 0.22: Model of working memory. Adapted from Baddeley.[75]

in their heads. Research has uncovered a system known as the phonological (or articulatory) loop. This kind of memory can be thought of as being like a loop of tape. Sounds can be recorded onto this tape, overwriting the previous contents, as it goes around and around. An example of the functioning of this loop can be found, by trying to remember lists of words that vary by the length of time it takes to say them.

Table 0.14 contains lists of words; those at the top of the table contain a single syllable, those at the bottom multiple syllables. Readers should have no problems remembering a sequence of five single-syllable words, a sequence of five multi-syllable words should prove more difficult. As before, read each word slowly out loud.

Table 0.14: Words with either one or more than one syllable (and thus varying in the length of time taken to speak).

List 1	List 2	List 3	List 4	List 5
one	cat	card	harm	add
bank	lift	list	bank	mark
sit	able	inch	view	bar
kind	held	act	fact	few
look	mean	what	time	sum
ability	basically	encountered	laboratory	commitment
particular	yesterday	government	acceptable	minority
mathematical	department	financial	university	battery
categorize	satisfied	absolutely	meaningful	opportunity
inadequate	beautiful	together	carefully	accidental

It has been found that fast talkers have better short-term memory. The connection is the phonological loop. Short-term memory is not limited by the number of items that can be held. The limit is the length of sound this loop can store, about two seconds.[76] Faster talkers can represent more information in that two seconds than those who do not talk as fast.

An analogy between *phonological loop* and a loop of tape in a tape recorder, suggests the possibility that it might only be possible to extract information as it goes past a *read-out point*. A study by Sternberg[1294] looked at how information in the phonological loop could be accessed. Subjects were asked to hold a sequences of digits, for instance 4185, in memory. They were then asked if a particular digit was in the sequence being held. The time taken to respond yes/no was measured. Subjects were given sequences of different length to hold in memory. The results showed that the larger the number of digits subjects had to hold in memory, the longer it took them to reply (see Figure 0.23). The other result was that the time to respond was not affected by whether the answer was yes or no. It might be expected that a yes answer would enable the search to be terminated. This suggests that all digits were always being compared.

memory span

A study by Cavanagh[210] found that different kinds of information, held in memory, has different search response times (see Figure 0.24).

A good example of using the different components of working memory is mental arithmetic; for example, multiply 23 by 15 without looking at this page. The numbers to be multiplied can be held in the phonological

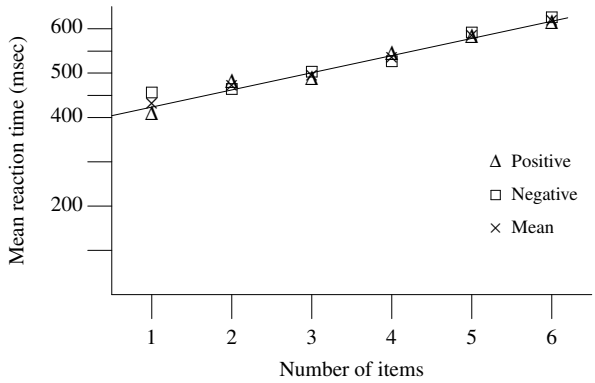


Figure 0.23: Judgment time (in milliseconds) as a function of the number of digits held in memory. Adapted from Sternberg.[1294]

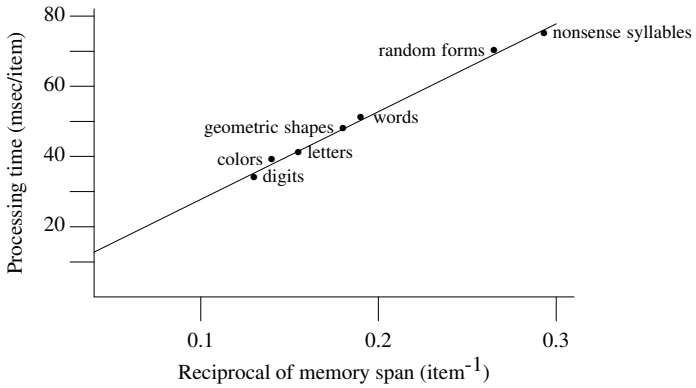


Figure 0.24: Judgment time (msec per item) as a function of the number of different items held in memory. Adapted from Cavanagh[210]

loop, while information such as carries and which two digits to multiple next can be held within the central executive. Now perform another multiplication, but this time look at the two numbers being multiplied (see margin for values) while performing the multiplication.

26
12
Two numbers to multiply.

While performing this calculation the visuo-spatial sketch pad can be used to hold some of the information, the values being multiplied. This frees up the phonological loop to hold temporary results, while the central executive holds positional information (used to decide which pairs of digits to look at). Carrying out a multiplication while being able to look at the numbers being multiplied seems to require less cognitive effort.

Recent research on working memory has begun to question whether it does have a capacity limit. Many studies have shown that people tend to organize items in memory in chunks of around four items. The role that attention plays in working memory, or rather the need for working memory in support of attention, has also come to the fore. It has been suggested that the focus of attention is capacity-limited, but that the other temporary storage areas are time-limited (without attention to rehearse them, they fade away). Cowan[296] proposed the following:

0 attention

1. The focus of attention is capacity-limited.
2. The limit in this focus averages about four chunks in normal adult humans.
3. No other mental faculties are capacity-limited, although some are limited by time and susceptibility to interference.

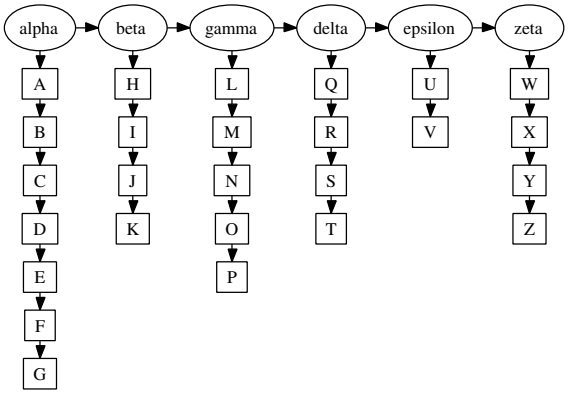


Figure 0.25: Semantic memory representation of alphabetic letters (the Greek names assigned to nodes by Klahr are used by the search algorithm and are not actually held in memory). Readers may recognize the structure of a nursery rhyme in the letter sequences. Derived from Klahr.[744]

4. Any information that is deliberately recalled, whether from a recent stimulus or from long-term memory, is restricted to this limit in the focus of attention.

Other studies[1022] have used the results from multiple tasks to distinguish the roles (e.g., storage, processing, supervision, and coordination) of different components of working memory.

memory chunking

Chunking is a technique commonly used by people to help them remember information. A chunk is a small set of items (4±1 is seen in many studies) having a common, strong, association with each other (and a much weaker one to items in other chunks). For instance, Wickelgren[1458] found that people's recall of telephone numbers is optimal if numbers are grouped into chunks of three digits. An example from random-letter sequences is *fbicbsibmirs*. The trigrams (*fbi*, *cbs*, *ibm*, *irs*) within this sequence of 12 letters are well-known acronyms. A person who notices this association can use it to aid recall. Several theoretical analyses of memory organizations have shown that chunking of items improves search efficiency ([360] optimal chunk size 3–4), ([880] number items at which chunking becomes more efficient than a single list, 5–7).

An example of chunking of information is provided by a study performed by Klahr, Chase, and Lovelace[744] who investigated how subjects stored letters of the alphabet in memory. Through a series of time-to-respond measurements, where subjects were asked to name the letter that appeared immediately before or after the presented probe letter, they proposed the alphabet-storage structure shown in Figure 0.25. They also proposed two search algorithms that described the process subjects used to answer the before/after question.

feeling of knowing

One of the characteristics of human memory is that it has knowledge of its own knowledge. People are good at judging whether they know a piece of information or not, even if they are unable to recall that information at a particular instant. Studies have found that so-called *feeling of knowing* is a good predictor of subsequent recall of information (see Koriat[765] for a discussion and a model).

working memory information representation

phonology 792

Several models of working memory are based on it only using a phonological representation of information. Any semantic effects in short-term memory come from information recalled from long-term memory. However, a few models of short-term memory do include a semantic representation of information (see Miyake and Shah[947] for detailed descriptions of all the current models of working memory, and Baddeley for a comprehensive review[71]).

A study by Hambrick and Engle[537] asked subjects to remember information relating to baseball games. The subjects were either young, middle age, or old adult who knew little about baseball or were very knowledgeable about baseball. The largest factor (54.9%) in the variance of subject performance was expertise, with working memory capacity and age making up the difference.

Source code constructs differ in their likelihood of forming semantically meaningful chunks. For instance,

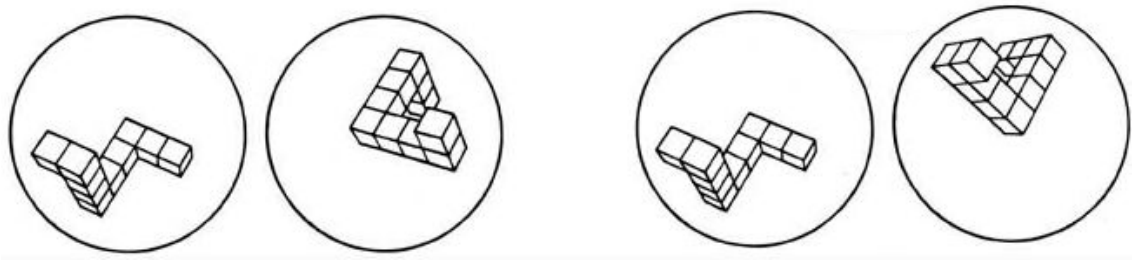


Figure 0.26: One of the two pairs are rotated copies of each other.

the ordering of a sequence of statements is often driven by the operations performed by those statements, while the ordering of parameters is often arbitrary.

Declarative memory is a long-term memory (information may be held until a person dies) that has a huge capacity (its bounds are not yet known) and holds information on facts and events (declarative knowledge is discussed elsewhere). Two components of declarative memory of interest to the discussion here are episodic and semantic memory. Episodic memory[72] is a past-oriented memory system concerned with *remembering*, while semantic memory is a present-oriented memory system concerned with *knowing*.

memory semantic memory episodic
0 declarative knowledge

Having worked on a program, a developer may remember particular sections of source code through their interaction with it (e.g., deducing how it interacted with other source code, or inserting traces to print out values of objects referenced in the code). After working on the same program for an extended period of time, a developer is likely to be able to recall information about it without being able to remember exactly when they learned that information.[565]

16.2.2.1 Visual manipulation

developer visual manipulation

How are visual images held in the brain? Are they stored directly in some way (like a bitmap), or are they held using an abstract representation (e.g., a list of objects tagged with their spatial positions). A study performed by Shepard[1226] suggested the former. He showed subjects pairs of figures and asked them if they were the same. Some pairs were different, while others were the same but had been rotated relative to each other. The results showed a linear relationship between the angle of rotation (needed to verify that two objects were the same) and the time taken to make a matching comparison. Readers might like to try there mind at rotating the pairs of images in Figure 0.26 to find out if they are the same.

Kosslyn[767] performed a related experiment. Subjects were shown various pictures and asked questions about them. One picture was of a boat. Subjects were asked a question about the front of the boat and then asked a question about the rear of the boat. The response time, when the question shifted from the front to the rear of the boat, was longer than when the question shifted from one about portholes to one about the rear. It was as if subjects had to scan their image of the boat from one place to another to answer the questions.

A study by Presson and Montello[1122] asked two groups of subjects to memorize the locations of objects in a room. Both groups of subjects were then blindfolded and asked to point to various objects. The results showed their performance to be reasonably fast and accurate. Subjects in the first group were then asked to imagine rotating themselves 90°, then they were asked to point to various objects. The results showed their performance to be much slower and less accurate. Subjects in the second group were asked to actually rotate 90°; while still blindfolded, they were then asked to point to various objects. The results showed that the performance of these subjects was as good as before they rotated. These results suggest that mentally keeping track of the locations of objects, a task that many cognitive psychologists would suspect as being cognitive and divorced from the body, is in fact strongly affected by literal body movements (this result is more evidence for *the embodied mind* theory[1411] of the human mind).

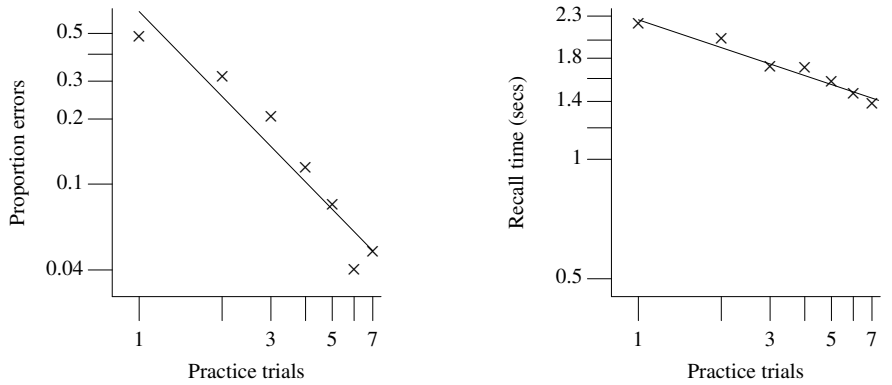


Figure 0.27: Proportion of errors (left) and time to recall (right) for recall of paired associate words (log scale). Based on Anderson.[32]

16.2.2.2 Longer term memories

People can store large amounts of information for long periods of time in their long-term memory. Landauer[801] attempts to estimate the total amount of learned information in LTM. Information written to LTM may not be held there for very long (storage), or it may be difficult to find (retrieval). This section discusses storage and retrieval of information in LTM.

One of the earliest memory research results was that practicing an item, after it had been learned, improves performance of recall at a later time (first published by Ebbinghaus in 1885, and reprinted several times since[374]). The relationship between practice, P, and time, T, to recall has been found to follow a power law $T = aP^b$ (where a and b are constants). This relationship has become known as *the power law of learning*. A similar relationship has been found for error rates— more practice, fewer errors.

How is information stored in LTM? The brain contains neurons and synapses; information can only be represented as some kind of change in their state. The term *memory trace* is used to describe this changed state, representing the stored information. Accessing an information item in LTM is thought to increase the strength of its associated *memory trace* (which could mean that a stronger signal is returned by subsequent attempts at recall, or that the access path to that information is smoothed; nobody knows yet).

Practice is not the only way of improving recall. How an item has been studied, and its related associations, can affect how well it is recalled later. The meaning of information to the person doing the learning, so-called *depth of processing*, can affect their recall performance. Learning information that has a meaning is thought to create more access methods to its storage location(s) in LTM.

synonym 792

The *generation effect* refers to the process whereby people are involved in the generation of the information they need to remember. A study by Slamecka and Graf[1253] asked subjects to generate a synonym, or rhyme, of a target word that began with a specified letter. For instance, generate a synonym for *sea* starting with the letter *o* (e.g., *ocean*). The subjects who had to generate the associated word showed a 15% improvement in recall, compared to subjects who had simply been asked to read the word pair (e.g., *sea–ocean*).

memory information elaboration

An example of the effect of additional, meaningful information was provided by a study by Bradshaw and Anderson.[150] Subjects were given information on famous people to remember. For instance, one group of subjects was told:

> Newton became emotionally unstable and insecure as a child

while other groups were given two additional facts to learn. These facts either elaborated on the first sentence or were unrelated to it:

Newton became emotionally unstable and insecure as a child
Newton's father died when he was born
Newton's mother remarried and left him with his grandfather

After a delay of one week, subjects were tested on their ability to recall the target sentence. The results showed that subjects percentage recall was higher when they had been given two additional sentences, that elaborated on the first one (the performance of subjects given related sentences being better than those given unrelated ones). There was no difference between subjects, when they were presented with the original sentence and asked if they recognized it.

The preceding studies involved using information that had a verbal basis. A study by Standing, Conezio, and Haber[1280] involved asking subjects to remember visual information. The subjects were shown 2,560 photographs for 10 seconds each (640 per day over a 4-day period). On each day, one hour after being shown the pictures, subjects were shown a random sample of 70 pairs of pictures (one of which was in the set of 640 seen earlier). They had to identify which of the pair they had seen before. Correct identification exceeded 90%. This and other studies have confirmed people's very good memory for pictures.

16.2.2.3 Serial order

memory
serial lists

The order in which items or events occur is often important when comprehending source code. For instance, the ordering of a function's parameters needs to be recalled when passing arguments, and the order of statements within the source code of a function specifies an order of events during program execution. Two effects are commonly seen in human memory recall performance:

1. The *primacy effect* refers to the better recall performance for items at the start of a list. primacy effect memory
2. The *recency effect* refers to the better recall performance for items at the end of a list. recency effect memory

A number of models have been proposed to explain people's performance in the serial list recall task. Henson[563] describes the *start–end model*.

16.2.2.4 Forgetting

forgetting

While people are unhappy about the fact that they forget things, never forgetting anything may be worse. The Russian mnemonist Shereshevskii found that his ability to remember everything, cluttered up his mind.[878] Having many similar, not recently used, pieces of information matching during a memory search would be counterproductive; forgetting appears to be a useful adaptation. For instance, a driver returning to a car wants to know where it was last parked, not the location of all previous places where it was parked. Anderson and Milson[36] proposed that human memory is optimized for information retrieval based on the statistical properties of information use, in people's everyday lives; their work was based on a model developed by Burrell[185] (who investigated the pattern of book borrowings in several libraries; which were also having items added to their stock). The rate at which the mind forgets seems to mirror the way that information tends to lose its utility in the real world over time.

It has only recently been reliably established[1185] that forgetting, like learning, follows a power law (the results of some studies could be fitted using exponential functions). The general relationship between the retention of information, R, and the time, T, since the last access has the form $R = aD^{-b}$ (where a and b are constants). It is known as the *power law of forgetting*. The constant a depends on the amount of initial learning. A study by Bahrick[80] (see Figure 0.28) looked at subjects' retention of English–Spanish vocabulary (the drop-off after 25 years may be due to physiological deterioration[33]).

The following are three theories of how forgetting occurs:

1. Memory traces simply fade away.
2. Memory traces are disrupted or obscured by newly formed memory traces created by new information being added to memory.

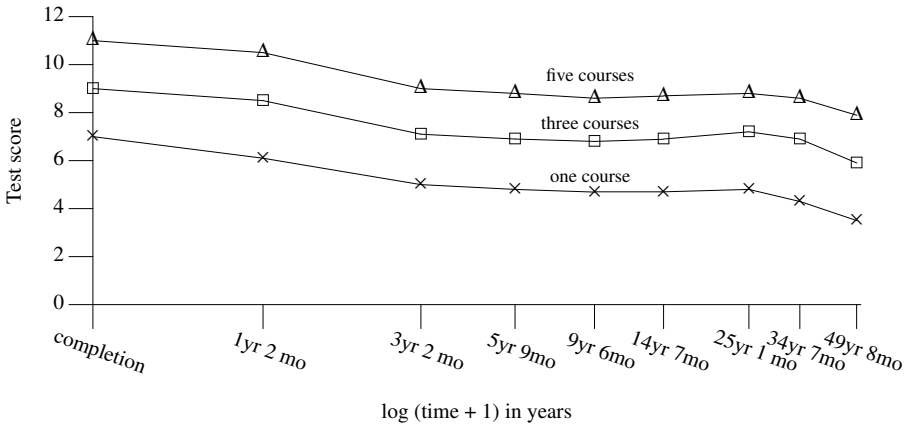


Figure 0.28: Effect of level of training on the retention of recognition of English–Spanish vocabulary. Adapted from Bahrick.[80]

3. The retrieval cues used to access memory traces are lost.

The process of learning new information is not independent of already-learned information. There can be mutual inference between the two items of information. The interference of old information, caused by new information, is known as *retroactive interference*. It is not yet known whether the later information weakens the earlier information, or whether it is simply stronger and overshadows access to the earlier information. The opposite effect of retroactive interference is *proactive interference*. In this case, past memories interfere with more recent ones.

Table 0.15 and Table 0.16 (based on Anderson[34]) are examples of the word-pair association tests used to investigate interference effects. Subjects are given a single pair of words to learn and are tested on that pair only (in both tables, Subject 3 is the control). The notation $A \Rightarrow B$ indicates that subjects have to learn to respond with B when given the cue A. An example of a word-pair is *sailor–tipsy*. The Worse/Better comparison is against the performance of the control subjects.

Table 0.15: Proactive inhibition. The third row indicates learning performance; the fifth row indicates recall performance, relative to that of the control. Based on Anderson.[34]

Subject 1	Subject 2	Subject 3
Learn A⇒B	Learn C⇒D	Rest
Learn A⇒D	Learn A⇒B	Learn A⇒D
Worse	Better	
Test A⇒D	Test A⇒D	Test A⇒D
Worse	Worse	

Table 0.16: Retroactive inhibition. The fourth row indicates subject performance relative to that of the control. Based on Anderson.[34]

Subject 1	Subject 2	Subject 3
Learn A⇒B	Learn A⇒B	Learn A⇒B
Learn A⇒D	Learn C⇒D	Rest
Test A⇒B	Test A⇒B	Test A⇒B
Much worse	Worse	

The general conclusion from the, many, study results is that interference occurs in both learning and recall

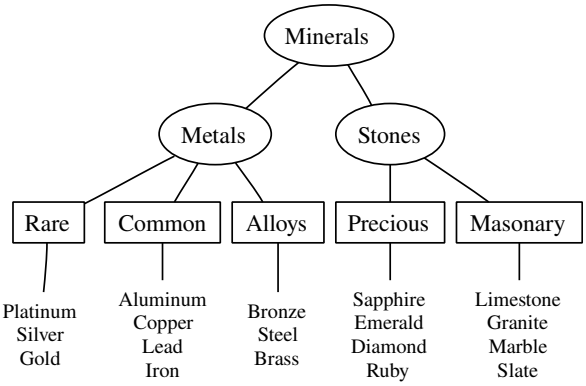


Figure 0.29: Words organized according to their properties— the *minerals* conceptual hierarchy. Adapted from Bower, Clark, Lesgold, and Winzenz.[147]

when there are multiple associations for the same item. The improvement in performance of subjects in the second category, of proactive inhibition, is thought to occur because of a practice effect.

16.2.2.5 Organized knowledge

Information is not stored in people's LTM in an unorganized form (for a detailed discussion, see[34]). This section provides a brief discussion of the issues. More detailed discussions are provided elsewhere in the specific cases that apply to reading and writing source code.

developers
organized
knowledge
792 identifier
memorability

Whenever possible, the coding guidelines given in this book aim to take account of the abilities and limitations that developers have. An example of how it is possible to use an ability of the mind (organizing information in memory) to overcome a limitation (information held in LTM becoming inaccessible) is provided by the following demonstration.

Readers might like to try remembering words presented in an organized form and as a simple list. Read the words in Figure 0.29 out loud, slowly and steadily. Then try to recall as many as possible. Then repeat the process using the words given below. It is likely that a greater number of words will be recalled from the organized structure. The words in the second list could be placed into the same structure as the first list, instead they appear in a random order.

```
      pine elm pansy garden wild banyan plants
   delphinium conifers dandelion redwood palm ash
violet daisy tropical chestnut flowers spruce lupin
   buttercup trees deciduous mango willow rose
```

Familiarity with information being learned and recalled can also make a difference. Several studies have shown that experts perform better than non-experts in remembering information within their domain of expertise. For instance, McKeithen, Reitman, Ruster, and Hirtle[917] measured developers' ability to memorize program source code. Subjects were presented with two listings; one consisted of a sequence of lines that made up a well-formed program, the other contained the same lines but the order in which they appeared on the listing had been randomized. Experienced developers (more than 2,000 hr of general programming and more than 400 hr experience with the language being used in the experiment) did a much better job at recalling source lines from the listing that represented a well-formed program and inexperienced developers. Both groups did equally well in recalling lines from the randomized listing. The experiments also looked at how developers remembered lists of language keywords they were given. How the information was organized was much more consistent across experienced developers than across inexperienced developers (experienced developers also had a slightly deeper depth of information chunking, 2.26 vs. 1.88).

16.2.2.6 Memory accuracy

Until recently experimental studies of memory have been dominated by a quantity-oriented approach. Memory was seen as a storehouse of information and is evaluated in terms of how many items could be successfully retrieved. The issue of accuracy of response was often ignored. This has started to change and there has been a growing trend for studies to investigate accuracy.[766] Coding guidelines are much more interested in factors that affect memory accuracy than those, for instance, affecting rate of recall. Unfortunately, some of the memory studies described in this book do not include information on error rates.

16.2.2.7 Errors caused by memory overflow

developer errors memory overflow

Various studies have verified that limits on working memory can lead to an increase in a certain kind of error when performing a complex task. Byrne and Bovair[188] looked at postcompletion errors (an example of this error is leaving the original in the photocopy machine after making copies, or the ATM card in the machine after withdrawing money) in complex tasks. A task is usually comprised of several goals that need to be achieved. It is believed that people maintain these goals using a stack mechanism in working memory. Byrne and Bovair were able to increase the probability of subjects making postcompletion errors in a task assigned to them. They also built a performance model that predicted postcompletion errors that were consistent with those seen in the experiments.

conditional statement 1739

The possible impact of working memory capacity-limits in other tasks, related to reading and writing source code, is discussed elsewhere. However, the complexity of carrying out studies involving working memory should not be underestimated. There can be unexpected interactions from many sources. A study by Lemaire, Abdi, and Fayol[833] highlighted the complexity of trying to understand the affects of working memory capacity limitations. The existing models of the performance of simple arithmetic operations, involve an interrelated network in long-term memory (built during the learning of arithmetic facts, such as the multiplication table, and reinforced by constant practice). Lemaire et al. wanted to show that simple arithmetic also requires working memory resources.

To show that working memory resources were required, they attempted to overload those resources. Subjects were required to perform another task at the same time as answering a question involving simple arithmetic (e.g., $4 + 8 = 12$, true or false?). The difficulty of the second task varied between experiments. One required subjects to continuously say the word *the*, another had them continuously say the letters *abcdef*, while the most difficult task required subjects to randomly generate letters from the set *abcdef* (this was expected to overload the central executive system in working memory).

phonological loop

The interesting part of the results (apart from confirming the authors' hypothesis that working memory was involved in performing simple arithmetic) was how the performances varied depending on whether the answer to the simple arithmetic question was true or false. The results showed that performance for problems that were true was reduced when both the phonological loop and the central executive were overloaded, while performance on problems that were false was reduced when the central executive was overloaded.

A conditional expression requires that attention be paid to it if a developer wants to know under what set of circumstances it is true or false. What working memory resources are needed to answer this question; does keeping the names of identifiers in the phonological loop, or filling the visuo-spatial sketch pad (by looking at the code containing the expression) increase the resources required; does the semantics associated with identifiers or conditions affect performance? Your author does not know the answers to any of these questions but suspects that these issues, and others, are part of the effort cost that needs to be paid in extracting facts from source code.

16.2.2.8 Memory and code comprehension

As the results from the studies just described show, human memory is far from perfect. What can coding guidelines do to try to minimize potential problems caused by these limitations? Some authors of other coding guideline documents have obviously heard of Miller's[935] 7±2 paper (although few seem to have read it), often selecting five as the maximum bound on the use of some constructs.[684] However, the effects of working memory capacity-limits cannot be solved by such simple rules. The following are some of the many issues that need to be considered:

- What code is likely to be written as a consequence of a guideline recommendation that specifies some limit on the use of a construct? Would following the guideline lead to code that was more difficult to comprehend?
- Human memory organizes information into related chunks (which can then be treated as a single item) multiple chunks may in turn be grouped together, forming a structured information hierarchy. The visibility of this structure in the visible source may be beneficial.
- There are different limits for different kinds of information.
- All of the constructs in the source can potentially require working memory resources. For instance, identifiers containing a greater number of syllables consume more resources in the phonological loop.

792 identifier cognitive resource usage

There has been some research on the interaction between human memory and software development. For instance, Altmann[18] built a computational process model based on SOAR, and fitted it to 10.5 minutes of programmer activity (debugging within an `emacs` window). The simulation was used to study the memories, called near-term memory by Altmann, built up while trying to solve a problem. However, the majority of studies discussed in this book are not directly related to reading and writing source code (your author has not been able to locate many). They can, at best, be used to provide indicators. The specific applications of these results occur throughout this book. They include reducing interference between information chunks and reducing the complexity of reasoning tasks.

792 identifier syntax
1739 selection statement syntax

16.2.2.9 Memory and aging

A study by Swanson[1322] investigated how various measures of working memory varied with the age of the subject. The results from diverse working memory tasks were reasonably intercorrelated. The following are the general conclusions:

memory ageing

- Age-related differences are better predicted by performance on tasks that place high demands on accessing information or maintaining old information in working memory than on measures of processing efficiency.
- Age-related changes in working memory appear to be caused by changes in a general capacity system.
- Age-related performance for both verbal and visuo-spatial working memory tasks showed similar patterns of continuous growth that peak at approximately age 45.

0 visuo-spatial memory

16.2.3 Attention

Attention is a limited resource provided by the human mind. It has been proposed that the age we live in is not the information age, but the attention age.[321] Viewed in resource terms there is often significantly more information available to a person than attention resources (needed to process it). This is certainly true of the source code of any moderately large application.

attention

Much of the psychology research on attention has investigated how inputs from our various senses handled. It is known that they operate in parallel and at some point there is a serial bottleneck, beyond which point it is not possible to continue processing input stimuli in parallel. The point at which this bottleneck occurs is a continuing subject of debate. There are early selection theories, late selection theories, and theories that combine the two.[1060] In this book, we are only interested in the input from one sense, the eyes. Furthermore, the scene viewed by the eyes is assumed to be under the control of the viewer. There are no objects that spontaneously appear or disappear; the only change of visual input occurs when the viewer turns a page or scrolls the source code listing on a display.

Read the bold print in the following paragraph:

Somewhere **Among** hidden **the** in **most** the **spectacular** Rocky Mountains **cognitive** near **abilities** Central City **is** Colorado **the** an **ability** old **to** miner **select** hid **one** a **message** box **from** of **another.** gold. **We** Although **do** several **this** hundred **by** people **focusing** have **our** looked **attention** for **on** it, **certain** they **cues** have **such** not **as** found **type** it **style**.

What do you remember from the regular, non-bold, text? What does this tell you about selective attention?

People can also direct attention to their internal thought processes and memories. Internal thought processes are the main subject of this section. The issue of automatization (the ability to perform operations automatically after a period of training) is also covered; visual attention is discussed elsewhere.

automatization 0

Ideas and theories of attention and conscious thought are often intertwined. While of deep significance, these issues are outside the scope of this book. The discussion in this section treats attention as a resource available to a developer when reading and writing source code. We are interested in knowing the characteristics of this resource, with a view to making the best use of the what is available. Studies involving attention have looked at capacity limits, the cost of changes of attention, and why some thought-conscious processes require more effort than others.

The following are two attention resource theories:

- *The single-capacity theory.* This proposes that performance depends on the availability of resources; more information processing requires more resources. When people perform more than one task at the same time, the available resources per task is reduced and performance decreases.
- *The multiple-resource theory.* This proposes that there are several different resources. Different tasks can require different resources. When people perform more than one task at the same time, the effect on the response for each task will depend on the extent to which they need to make use of the same resource at the same time.

Many of the multiple-resource theory studies use different sensory input tasks; for instance, subjects are required to attend to a visual and an audio channel at the same time. Reading source code uses a single sensory input, the eyes. However, the input is sufficiently complex that it often requires a great deal of thought. The extent to which code reading *thought* tasks are sufficiently different that they will use different cognitive resources is unknown. Unless stated otherwise, subsequent discussion of attention will assume that the tasks being performed, in a particular context, call on the same resources.

memory 0 developer

As discussed previously, the attention, or rather the focus of attention is believed to be capacity-limited. Studies suggest that this limit is around four chunks.[296] Studies[1420] have also found that attention performance has an age-related component.

power law of learning

Power law of learning

Studies have found that nearly every task that exhibits a practice effect follows the *power law of learning*; which has the form:

$$RT = a + bN^{-c} \tag{0.21}$$

where RT is the response time; N is the number of times the task has been performed; and a, b, and c are constants. There were good theoretical reasons for expecting the equation to have an exponential form (i.e., $a + be^{-cN}$); many of the experimental results could be fitted to such an equation. However, if chunking is assumed to play a part in learning, a power law is a natural consequence (see Newell[1008] for a discussion).

16.2.4 Automatization

automatization culture of C 0

Source code contains several frequently seen patterns of usage. Experienced developers gain a lot of experience writing (or rather typing in) these constructs. As experience is gained, developers learn to type in these constructs without giving much thought to what they are doing. This process is rather like learning to write at school; children have to concentrate on learning to form letters and the combination of letters that form a word. After sufficient practice, many words only need to be briefly thought before they appear on the page without conscious effort.

The *instance theory* of automatization[864] specifies that novices begin by using an algorithm to perform a task. As they gain experience they learn specific solutions to specific problems. These solutions are retrieved

from memory when required. Given sufficient experience, the solution to all task-related problems can be obtained from memory and the algorithmic approach, to that task, is abandoned. The underlying assumptions of the theory are that encoding of problem and solution in memory is an unavoidable consequence of attention. Attending to a stimulus is sufficient to cause it to be committed to memory. The theory also assumes that retrieval of the solution from memory is an unavoidable consequence of attending to the task (this retrieval may not be successful, but it occurs anyway). Finally, each time the task is encountered (the instances) it causes encoding, storing, and retrieval, making it a learning-based theory.

Automatization (or automaticity) is an issue for coding guidelines in that many developers will have learned to use constructs whose use is recommended against. Developers' objections to having to stop using constructs that they know so well, and having to potentially invest in learning new techniques, is something that management has to deal with.

16.2.5 Cognitive switch

cognitive switch

Some cognitive processes are controlled by a kind of *executive* mechanism. The nature of this executive is poorly understood and its characteristics are only just starting to be investigated.[731] The process of comprehending source code can require switching between different tasks. Studies[959] have found that subjects responses are slower and more error prone immediately after switching tasks. The following discussion highlights the broader research results.

A study by Rogers and Monsell[1177] used the two tasks of classifying a letter as a consonant or vowel, and classifying a digit as odd or even. The subjects were split into three groups. One group was given the latter classification task, the second group the digit classification task, and the third group had to alternate (various combinations were used) between letter and digit classification. The results showed that having to alternate tasks slowed the response times by 200 to 250 ms and the error rates went up from 2% to 3% to 6.5% to 7.5%. A study by Altmann[19] found that when the new task shared many features in common with the previous task (e.g., switching from classifying numbers as odd or even, to classifying them as less than or greater than five) the memories for the related tasks interfered, causing a reduction in subject reaction time and an increase in error rate.

The studies to date have suggested the following conclusions:[406]

- When it occurs the alternation cost is of the order of a few hundred milliseconds, and greater for more complex tasks.[1187]
- When the two tasks use disjoint stimulus sets, the alternation cost is reduced to tens of milliseconds, or even zero. For instance, the tasks used by Spector and Biederman[1270] were to subtract three from Arabic numbers and name antonyms of written words.

792 antonym

- Adding a cue to each item that allows subjects to deduce which task to perform reduces the alternation cost. In the Spector and Biederman study, they suffixed numbers with "+3" or "-3" in a task that required them to add or subtract three from the number.
- An alternation cost can be found in tasks having disjoint stimulus sets when those stimulus sets occurred in another pair of tasks that had recently been performed in alternation.

These conclusions raise several questions in a source code reading context. To what extent do different tasks involve different stimulus sets and how prominent must a cue be (i.e., is the 0x on the front of a hexadecimal number sufficient to signal a change of number base)? These issues are discussed elsewhere under the C language constructs that might involve cognitive task switches.

884 character constant value
945 bitwise operators

Probably the most extreme form of cognitive switch is an external interruption. In some cases, it may be necessary for developers to perform some external action (e.g., locating a source file containing a needed definition) while reading source code. Latorella[813] discusses the impact of interruptions on the performance of flight deck personnel (in domains where poor performance in handling interruptions can have fatal consequences), and McFarlane[915] provides a human-computer interruption taxonomy.

16.2.6 Cognitive effort

cognitive effort

Why do some mental processes seem to require more mental effort than others? Why is effort an issue in mental operations? The following discussion is based on Chapter 8 of Pashler.[1060]

One argument is that mental effort requires energy, and the body's reaction to concentrated thinking is to try to conserve energy by creating a sense of effort. Studies of blood flow show that the brain accounts for 20% of heart output, and between 20% to 25% of oxygen and glucose requirements. But, does concentrated thinking require a greater amount of metabolic energy than sitting passively? The answer from PET scans of the brain appears to be no. In fact the energy consumption of the visual areas of the brain while watching television are higher than the consumption levels of those parts of the brain associated with *difficult thinking*.

Another argument is that the repeated use of neural systems produces a temporary reduction in their efficiency. A need to keep these systems in a state of readiness (fight or flight) could cause the sensation of mental effort. The results of some studies are not consistent with this repeated use argument.

The final argument, put forward by Pashler, is that *difficult thinking* puts the cognitive system into a state where it is close to failing. It is the internal recognition of a variety of signals of impending cognitive failure that could cause the feeling of mental effort.

At the time of this writing there is no generally accepted theory of the root cause of cognitive effort. It is a recognized effect and developers' reluctance to experience it is a factor in the specification some of the guideline recommendations.

What are the components of the brain that are most likely to be resource limited when performing a source code comprehension task? Source code comprehension involves many of the learning and problem solving tasks that students encounter in the class room. Studies have found a significant correlation between the working memory requirements of a problem and students' ability to solve it[1279] and teenagers academic performance in mathematics and science subjects (but not English).[474]

Most existing research has attempted to find a correlation between a subjects learning and problem solving performance and the capacity of their working memory.[259] Some experiments have measured subjects recall performance, after performing various tasks. Others have measured subjects ability to make structure the information they are given into a form that enables them to answer questions about it[534] (e.g., who met who in "The boy the girl the man saw met slept.").

cognitive load

Cognitive load might be defined as the total amount of mental activity imposed on working memory at any instant of time. The cognitive effort needed to solve a problem being the sum of all the cognitive loads experienced by the person seeking the solution.

$$Cognitive\ effort = \sum_{i=i}^{t} Cognitive\ load_i \tag{0.22}$$

Possible techniques for reducing the probability that a developers working memory capacity will be exceeded during code comprehension include:

memory 0 chunking

- organizing information into chunks that developers are likely to recognize and have stored in their long-term memory,

identifier 792 cognitive resource usage

- minimizing the amount of information that developers need to simultaneously keep in working memory during code comprehension (i.e., just in time information presentation),
- minimizing the number of relationships between the components of a problem that need to be considered (i.e., break it up into smaller chunks that can be processed independently of each other). Algorithms based on database theory and neural networks[534] have been proposed as a method of measuring the *relational complexity* of a problem.

16.2.7 Human error

The discussion in this section has been strongly influenced by *Human Error* by Reason.[1149] Models of errors made by people have been broken down, by researchers, into different categories.

developer errors

Table 0.17: Main failure modes for skill-based performance. Adapted from Reason.[1149]

Inattention	Over Attention
Double-capture slips	Omissions
Omissions following interruptions	Repetitions
Reduced intentionality	Reversals
Perceptual confusions	
Interference errors	

- *Skill*-based errors (see Table 0.17) result from some failure in the execution and/or the storage stage of an action sequence, regardless of whether the plan which guided when was adequate to achieve its objective. Those errors that occur during execution of an action are called *slips* and those that occur because of an error in memory are called *lapses*.

- *Mistakes* can be defined as deficiencies or failures in the judgmental and/or inferential processes involved in the selection of an objective or in the specification of the means to achieve it, irrespective of whether the actions directed by this decision-scheme run according to plan. *Mistakes* are further categorized into one of two kinds— *knowledge-based* mistakes (see Table 0.18) mistakes and *rule based* mistakes (see Table 0.19).

Table 0.18: Main failure modes for knowledge-based performance. Adapted from Reason.[1149]

Knowledge-based Failure Modes
Selectivity
Workspace limitations
Out of, sight out of mind
Confirmation bias
Overconfidence
Biased reviewing
Illusory correlation
Halo effects
Problems with causality
Problems with complexity
Problems with delayed feed-back
Insufficient consideration of processes in time
Difficulties with exponential developments
Thinking in causal series not causal nets (unaware of side-effects of action)
Thematic vagabonding (flitting from issue to issue)
Encysting (lingering in small detail over topics)

This categorization can be of use in selecting guideline recommendations. It provides a framework for matching the activities of developers against existing research data on error rates. For instance, developers would make skill-based errors while typing into an editor or using cut-and-paste to move code around.

Table 0.19: Main failure modes for rule-base performance. Adapted from Reason.[1149]

Misapplication of Good Rules	Application of Bad Rules
First exceptions	Encoding deficiencies
Countersigns and nosigns	Action deficiencies
Information overload	Wrong rules
Rule strength	Inelegant rules
General rules	Inadvisable rules
Redundancy	
Rigidity	

16.2.7.1 Skill-based mistakes

The consequences of possible skill-based mistakes may result in a coding guideline being created. However, by their very nature these kinds of mistakes cannot be directly recommended against. For instance, mistypings of identifier spellings leads to a guideline recommendation that identifier spellings differ in more than one significant character. A guideline recommending that identifier spellings not be mistyped being pointless.

identifier typed form 792.5

Information on instances of this kind of mistake can only come from experience. They can also depend on development environments. For instance, cut-and-paste mistakes may vary between use of line-based and GUI-based editors.

16.2.7.2 Rule-based mistakes

rule-base mistakes

Use of rules to perform a task (a rule-based performance) does not imply that if a developer has sufficient expertise within the given area that they no longer need to expend effort thinking about it (a knowledge-based performance), only that a rule has been retrieved, from the memory, and a decision made to use it (rending a knowledge-based performance).

The starting point for the creation of guideline recommendations intended to reduce the number of rule-based mistakes, made by developers is an extensive catalog of such mistakes. Your author knows of no such catalog. An indication of the effort needed to build such a catalog is provided by a study of subtraction mistakes, done by VanLehn.[1409] He studied the mistakes made by children in subtracting one number from another, and built a computer model that predicted many of the mistakes seen. The surprising fact, in the results, was the large number of diagnosed mistakes (134 distinct diagnoses, with 35 occurring more than once). That somebody can write a 250-page book on subtraction mistakes, and the model of procedural errors built to explain them, is an indication that the task is not trivial.

Holland, Holyoak, Nisbett, and Thagard[585] discuss the use of rules in solving problems by induction and the mistakes that can occur through different rule based performances.

16.2.7.3 Knowledge-based mistakes

Mistakes that occur when people are forced to use a knowledge-based performance have two basic sources: bounded rationality and an incomplete or inaccurate mental model of the problem space.

bounded rationality 0

A commonly used analogy of knowledge-based performances is that of a beam of light (working memory) that can be directed at a large canvas (the mental map of the problem). The direction of the beam is partially under the explicit control of its operator (the human conscious). There are unconscious influences pulling the beam toward certain parts of the canvas and avoiding other parts (which may, or may not, have any bearing on the solution). The contents of the canvas may be incomplete or inaccurate.

People adopt a variety of strategies, or heuristics, to overcome limitations in the cognitive resources available to them to perform a task. These heuristics appear to work well in the situations encountered in everyday human life, especially so since they are widely used by large numbers of people who can share in a common way of thinking.

Reading and writing source code is unlike everyday human experiences. Furthermore, the reasoning methods used by the non-carbon-based processor that executes software are wholly based on mathematical logic, which is only one of the many possible reasoning methods used by people (and rarely the preferred one at that).

There are several techniques for reducing the likelihood of making knowledge-based mistakes. For instance, reducing the size of the canvas that needs to be scanned and acknowledging the effects of heuristics.

940 expressions
0 availability heuristic
0 representative heuristic

16.2.7.4 Detecting errors

The modes of control for both skill-based and rule-based performances are feed-forward control, while the mode for knowledge-based performances is feed-back control. Thus, the detection of any skill-based or rule-based mistakes tends to occur as soon as they are made, while knowledge-based mistakes tend to be detected long after they have been made.

There have been studies looking at how people diagnose problems caused by knowledge-based mistakes.[522] However, these coding guidelines are intended to provide advice on how to reduce the number of mistakes, not how to detect them once they have been made. Enforcement of coding guidelines to ensure that violations are detected is a very important issue.

0 guideline recommendation enforceable

16.2.7.5 Error rates

people error rates

There have been several studies of the quantity of errors made by people performing various tasks. It is relatively easy to obtain this information for tasks that involve the creation of something visible (e.g., written material, of a file on a computer). Obtaining reliable error rates for information that is read and stored (or not) in people's memory is much harder to obtain. The following error rates may be applicable to writing source code:

792 typing mistakes

- Touch typists, who are performing purely data entry:[903] with no error correction 4% (per keystroke), typing nonsense words (per word) 7.5%.
- Typists using a line-oriented word processor:[1203] 3.40% of (word) errors were detected and corrected by the typist while typing, 0.95% were detected and corrected during proofreading by the typist, and 0.52% were not detected by the typist.
- Students performing calculator tasks and table lookup tasks: per multipart calculation, per table lookup, 1% to 2%.[925]

16.2.8 Heuristics and biases

Heuristics and Biases

In the early 1970s Amos Tversky, Daniel Kahneman, and other psychologists[705] performed studies, the results of which suggested people reason and make decisions in ways that systematically violate (mathematical based) rules of rationality. These studies covered a broad range of problems that might occur under quite ordinary circumstances. The results sparked the growth of a very influential research program often known as the *heuristics and biases* program.

There continues to be considerable debate over exactly what conclusions can be drawn from the results of these studies. Many researchers in the heuristics and biases field claim that people lack the underlying rational competence to handle a wide range of reasoning tasks, and that they exploit a collection of simple heuristics to solve problems. It is the use of these heuristics that make them prone to non-normative patterns of reasoning, or biases. This position, sometimes called the *standard picture*, claims that the appropriate norms for reasoning must be derived from mathematical logic, probability, and decision theory. An alternative to the standard Picture is proposed by evolutionary psychology. These researchers hold that logic and probability are not the norms against which human reasoning performance should be measured.

0 evolutionary psychology

When reasoning about source code the appropriate norm is provided by the definition of the programming language used (which invariably has a basis in at least first order predicate calculus). This is not to say that probability theory is not used during software development. For instance, a developer may choose to make use of information on commonly occurring cases (such usage is likely to be limited to ordering by frequency or probability; Bayesian analysis is rarely seen).

What do the results of the heuristics and biases research have to do with software development, and do they apply to the kind of people who work in this field? The subjects used in these studies were not, at the time of the studies, software developers. Would the same results have been obtained if software developers

developer 0 mental characteristics

had been used as subjects? This question implies that developers' cognitive processes, either through training or inherent abilities, are different from those of the subjects used in these studies. The extent to which developers are susceptible to the biases, or use the heuristics, found in these studies is unknown. Your author assumes that they are guilty until proven innocent.

Another purpose for describing these studies is to help the reader get past the idea that people exclusively apply mathematical logic and probability in problem solving.

16.2.8.1 Reasoning

developer reasoning

Comprehending source code involves performing a significant amount of reasoning over a long period of time. People generally consider themselves to be good at reasoning. However, anybody who has ever written a program knows how many errors are made. These errors are often claimed, by the author, to be caused by any one of any number of factors, except poor reasoning ability. In practice people are good at certain kinds of reasoning problems (the kind seen in everyday life) and very poor at others (the kind that occur in mathematical logic).

The basic mechanisms used by the human brain, for reasoning, have still not been sorted out and are an area of very active research. There are those who claim that the mind is some kind of general-purpose processor, while others claim that there are specialized units designed to carry out specific kinds of tasks (such as solving specific kinds of reasoning problems). Without a general-purpose model of human reasoning, there is no more to be said in this section. Specific constructs involving specific reasoning tasks are discussed in the relevant sentences.

selection statement 1739 syntax
logical-AND-expression 1248 syntax
logical-OR-expression 1256 syntax

16.2.8.2 Rationality

developer rationality

Many of those who study software developer behavior (there is no generic name for such people) have a belief in common with many economists. Namely, that their subjects act in a rational manner, reaching decisions for well-articulated goals using mathematical logic and probability, and making use of all the necessary information. They consider decision making that is not based on these norms as being *irrational*.

bounded rationality

Deciding which decisions are the rational ones to make requires a norm to compare against. Many early researchers assumed that mathematical logic and probability were the norm against which human decisions should be measured. The term *bounded rationality*[1240] is used to describe an approach to problem solving performed when limited cognitive resources are available to process the available information. A growing number of studies[489] are finding that the methods used by people to make decisions and solve problems are often optimal, given the resources available to them. A good discussion of the issues, from a psychology perspective, is provided by Samuels, Stich and Faucher.[1195]

For some time a few economists have been arguing that people do not behave according to mathematical norms, even when making decisions that will affect their financial well-being.[914] Evidence for this heresy has been growing. If people deal with money matters in this fashion, how can their approach to software development fare any better? Your author takes the position, in selecting some of the guideline recommendations in this book, that developers' cognitive processes when reading and writing source are no different than at other times.

When reading and writing source code written in the C language, the rationality norm is defined in terms of the output from the C abstract machine. Some of these guideline recommendations are intended to help ensure that developers' comprehension of source agrees with this norm.

16.2.8.3 Risk asymmetry

risk asymmetry

The term *risk asymmetry* refers to the fact that people are *risk averse* when deciding between alternatives that have a positive outcome, but are *risk seeking* when deciding between alternatives that have a negative outcome.

Making a decision using uncertain information involves an element of risk; the decision may not be the correct one. How do people handle risk?

Kahneman and Tversky[709] performed a study in which subjects were asked to make choices about gaining or losing money. The theory they created, *prospect theory*, differed from the accepted theory of the day,

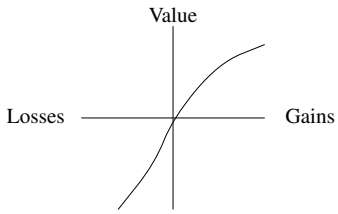


Figure 0.30: Relationship between subjective value to gains and to losses. Adapted from Kahneman.[709]

expected utility theory (which still has followers). Subjects were presented with the following problems:

> Problem 1: In addition to whatever you own, you have been given 1,000. You are now asked to choose between:
> A: Being given a further 1,000, with probability 0.5
> B: Being given a further 500, unconditionally
> Problem 2: In addition to whatever you own, you have been given 2,000. You are now asked to choose between:
> C: Loosing 1,000, with probability 0.5
> D: Loosing 500, unconditionally

The majority of the subjects chose B (84%) in the first problem, and C (69%) in the second. These results, and many others like them, show that people are risk averse for positive prospects and risk seeking for negative ones (see Figure 0.30).

In the following problem the rational answer, based on knowledge of probability, is E; however, 80% of subjects chose F.

> Problem 3: You are asked to choose between:
> E: Being given 4,000, with probability 0.8
> F: Being given 3,000, unconditionally

Kahneman and Tversky also showed that people's subjective probabilities did not match the objective probabilities. Subjects were given the following problems:

> Problem 4: You are asked to choose between:
> G: Being given 5,000, with probability 0.001
> H: Being given 5, unconditionally
> Problem 5: You are asked to choose between:
> I: Loosing 5,000, with probability 0.001
> J: Loosing 5, unconditionally

Most the subjects chose G (72%) in the first problem and J (83%) in the second.

Problem 4 could be viewed as a lottery ticket (willing to forego a small amount of money for the chance of wining a large amount), while Problem 5 could be viewed as an insurance premium (willingness to pay a small amount of money to avoid the possibility of having to pay out a large amount).

The decision weight given to low probabilities tends to be higher than that warranted by the evidence. The decision weight given to other probabilities tends to be lower than that warranted by the evidence (see Figure 0.31).

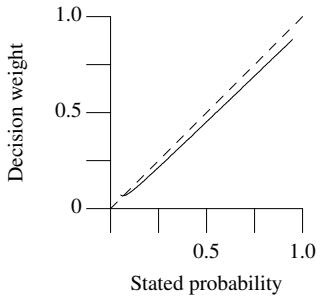


Figure 0.31: Possible relationship between subjective and objective probability. Adapted from Kahneman.[709]

16.2.8.4 Framing effects

framing effect

The framing effect occurs when alternative framings of what is essentially the same decision task cause predictably different choices.

Kahneman and Tversky[708] performed a study in which subjects were asked one of the following question:

> Imagine that the U.S. is preparing for the outbreak of an unusual Asian disease, which is expected to kill 600 people. Two alternative programs to combat the disease have been proposed. Assume that the exact scientific estimates of the consequences of the programs are as follows:
>
> If Program A is adopted, 200 people will be saved.
>
> If Program B is adopted, there is a one-third probability that 600 people will be saved and a two thirds probability that no people will be saved.
>
> Which of the two programs would you favor?

This problem is framed in terms of 600 people dying, with the option being between two programs that save lives. In this case subjects are risk averse with a clear majority, 72%, selecting Program A. For the second problem the same cover story was used, but subjects were asked to select between differently worded programs:

> If Program C is adopted, 400 people will die.
>
> If Program D is adopted, there is a one-third probability that nobody will die and two-thirds probability that 600 people will die.

In terms of their consequences Programs A and B are mathematically the same as C and D, respectively. However, this problem is framed in terms of no one dying. The best outcome would be to maintain this state of affairs. Rather than accept an unconditional loss, subjects become risk seeking with a clear majority, 78%, selecting Program D.

Even when subjects were asked both questions, separated by a few minutes, the same reversals in preference were seen. These results have been duplicated in subsequent studies by other researchers.

16.2.8.5 Context effects

context effects

The standard analysis of the decision's people make assumes that they are procedure-invariant; that is, assessing the attributes presented by different alternatives should always lead to the same one being selected. Assume, for instance, that in a decision task, a person chooses alternative X, over alternative Y. Any previous decisions they had made between alternatives similar to X and Y would not be thought to affect later decisions. Similarly, the addition of a new alternative to the list of available alternatives should not cause Y to be selected, over X.

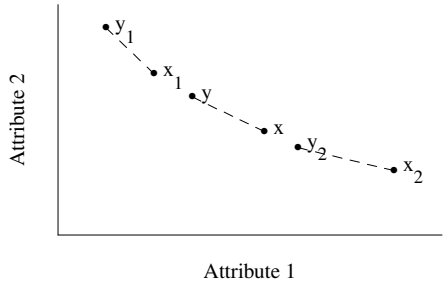


Figure 0.32: Text of background trade-off. Adapted from Tversky.[1377]

People will show procedure-invariance if they have well-defined values and strong beliefs. In these cases the appropriate values might be retrieved from a master list of preferences held in a person's memory. If preferences are computed using some internal algorithm, each time a person has to make a decision, then it becomes possible for context to have an affect on the outcome.

Context effects have been found to occur because of the prior history of subjects answering similar questions, background context, or because of presentation of the problem itself, local context. The following two examples are taken from a study by Tversky and Simonson.[1377]

To show that prior history plays a part in a subjects judgment, Tversky and Simonson split a group of subjects in two. The first group was asked to decide between the alternatives X_1 and Y_1, while the second group was asked to select between the options X_2 and Y_2. Following this initial choice all subjects were asked to chose between X and Y.

Table 0.20: Percentage of each alternative selected by subject groups S_1 and S_2. Adapted from Tversky.[1377]

Warranty	Price	S_1	S_2
X_1	\$85	12%	
Y_1	\$91	88%	
X_2	\$25		84%
Y_2	\$49		16%
X	\$60	57%	33%
Y	\$75	43%	67%

Subjects previously exposed to a decision where a small difference in price (see Table 0.20) (\$85 vs. \$91) was associated with a large difference in warranty (55,000 miles vs. 75,000 miles), were more likely to select the less-expensive tire from the target set (than those exposed to the other background choice, where a large difference in price was associated with a small difference in warranty).

In a study by Simonson and Tversky,[1243] subjects were asked to decide between two microwave ovens. Both were on sale at 35% off the regular price, at sale prices of \$109.99 and \$179.99. In this case 43% of the subjects selected the more expensive model. For the second group of subjects, a third microwave oven was added to the selection list. This third oven was priced at \$199.99, 10% off its regular price. The \$199.99 microwave appeared inferior to the \$179.99 microwave (it had been discounted down from a lower regular price by a smaller amount), but was clearly superior to the \$109.99 model. In this case 60% selected the \$179.99 microwave (13% chose the more expensive microwave). The presence of a third alternative had caused a significant number of subjects to switch the model selected.

16.2.8.6 Endowment effect

Studies have shown that losses are valued far more than gains. This asymmetry in the value assigned, by people, to goods can be seen in the endowment effect. A study performed Knetsch[749] illustrates this effect.

endowment effect
0 risk asymmetry

Subjects were divided into three groups. The first group of was given a coffee mug, the second group

was given a candy bar, and the third group was given nothing. All subjects were then asked to complete a questionnaire. Once the questionnaires had been completed, the first group was told that they could exchange their mugs for a candy bar, the second group that they could exchange their candy bar for a mug, while the third group was told they could decide between a mug or a candy bar. The mug and the candy bar were sold in the university bookstore at similar prices.

Table 0.21: Percentage of subjects willing to exchange what they had been given for an equivalently priced item. Adapted from Knetsch.[749]

Group	Yes	No
Give up mug to obtain candy	89%	11%
Give up candy to obtain mug	90%	10%

The decisions made by the third group, who had not been given anything before answering the questionnaire, were: mug 56%, candy 44%. This result showed that the perceived values of the mug and candy bar were close to each other.

The decisions made by the first and second groups (see Table 0.21) showed that they placed a higher value on a good they owned than one they did not own (but could obtain via a simple exchange).

The endowment effect has been duplicated in many other studies. In some studies, subjects required significantly more to sell a good they owned than they would pay to purchase it.

16.2.8.7 Representative heuristic

representative heuristic

The representative heuristic evaluates the probability of an uncertain event, or sample, by the degree to which it

- is similar in its essential attributes to the population from which it is drawn, and
- reflects the salient attributes of the process that generates it

given two events, X and Y. The event X is judged to be more probable than Y when it is more representative. The term *subjective probability* is sometimes used to describe these probabilities. They are subjective in the sense that they are created by the people making the decision. *Objective probability* is the term used to describe the values calculated from the stated assumptions, according to the axioms of mathematical probability.

base rate neglect

Selecting alternatives based on the representativeness of only some of their attributes can lead to significant information being ignored; in particular the nonuse of base-rate information provided as part of the specification of a problem.

Treating representativeness as an operator, it is a (usually) directional relationship between a family, or process M, and some instance or event X, associated with M. It can be defined for (1) a value and a distribution, (2) an instance and a category, (3) a sample and a population, or (4) an effect and a cause. These four basic cases of representativeness occur when (Tversky[1375]):

1. M is a family and X is a value of a variable defined in this family. For instance, the representative value of the number of lines of code in a function. The most representative value might be the mean for all the functions in a program, or all the functions written by one author.
2. M is a family and X is an instance of that family. For instance, the number of lines of code in the function *foo_bar*. It is possible for an instance to be a family. The *Robin* is an instance of the bird family and a particular individual can be an instance of the Robin family.
3. M is a family and X is a subset of M. Most people would agree that the population of New York City is less representative of the US than the population of Illinois. The criteria for representativeness in

a subset is not the same as for one instance. A single instance can represent the primary attributes of a family. A subset has its own range and variability. If the variability of the subset is small, it might be regarded as a category of the family, not a subset. For instance, the selected subset of the family birds might only include Robins. In this case, the set of members is unlikely to be regarded as a representative subset of the bird family.

4. M is a (causal) system and X is a (possible) instance generated by it. Here M is no longer a family of objects, it is a system for generating instances. An example would be the mechanism of tossing coins to generate instances of heads and tails.

16.2.8.7.1 Belief in the law of small numbers

law of small numbers

Studies have shown that people have a strong belief in what is known as the law of small numbers. This "law" might be stated as: "Any short sequence of events derived from a random process shall have the same statistical properties as that random process." For instance, if a fairly balanced coin is tossed an infinite number of times the percentage of heads seen will equal the percentage of tails seen. However, according to the law of small numbers, any short sequence of coin tosses will also have this property. Statistically this is not true, the sequences *HHHHHHHHHH* and *THHTHTTHTH* are equally probable, but one of them does not appear to be representative of a random sequence.

Readers might like to try the following problem.

> The mean IQ of the population of eighth graders in a city is *known* to be 100. You have selected a random sample of 50 children for a study of educational achievement. The first child tested has an IQ of 150.
>
> What do you expect the mean IQ to be for the whole sample?

Did you believe that because the sample of 50 children was randomly chosen from a large population, with a known property, that it would also have this property?; that is, the answer would be 100? The effect of a child with a high IQ being canceled out by a child with a very low IQ? The correct answer is 101; the known information, from which the mean should be calculated, is that we have 49 children with an estimated average of 100 and one child with a known IQ of 150.

16.2.8.7.2 Subjective probability

subjective probability

In a study by Kahneman and Tversky,[707] subjects were divided into two groups. Subjects in one group were asked the *more than* question, and those in the other group the *less than* question.

> An investigator studying some properties of a language selected a paperback and computed the average word-length in every page of the book (i.e., the number of letters in that page divided by the number of words). Another investigator took the first line in each page and computed the line's average word-length. The average word-length in the entire book is four. However, not every line or page has exactly that average. Some may have a higher average word-length, some lower.
>
> The first investigator counted the number of pages that had an average word-length of 6 or (more/less) and the second investigator counted the number of lines that had an average word-length of 6 or (more/less). Which investigator do you think recorded a larger number of such units (pages for one, lines for the other)?

Table 0.22: Percentage of subjects giving each answer. Correct answers are starred. Adapted from Kahneman.[707]

Choice	Less than 6	More than 6
The page investigator	20.8%*	16.3%
The line investigator	31.3%	42.9%*
About the same (i.e., within 5% of each other)	47.9%	40.8%

The results (see Table 0.22) showed that subjects judged equally representative outcomes to be equally likely, the size of the sample appearing to be ignored.

When dealing with samples, those containing the smaller number of members are likely to exhibit the largest variation. In the preceding case, the page investigator is using the largest sample size and is more likely to be closer to the average (4), which is less than 6. The line investigator is using a smaller sample of the book's contents and is likely to see a larger variation in measured word length (more than 6 is the correct answer here).

16.2.8.8 Anchoring

Anchoring

Answers to questions can be influenced by completely unrelated information. This was dramatically illustrated in a study performed by Tversky and Kahneman.[1374] They asked subjects to estimate the percentage of African countries in the United Nations. But, before stating their estimate, subjects were first shown an arbitrary number, which was determined by spinning a *wheel of fortune* in their presence. In some cases, for instance, the number 65 was selected, at other times the number 10. Once a number had been determined by the *wheel of fortune* subjects were asked to state whether the percentage of African countries in the UN was higher or lower than this number, and their estimate of the percentage. The median estimates were 45% of African countries for subjects whose *anchoring* number was 65, and 25% for subjects whose *anchoring* number was 10.

The implication of these results is that people's estimates can be substantially affected by a numerical *anchoring* value, even when they are aware that the anchoring number has been randomly generated.

16.2.8.9 Belief maintenance

belief maintenance

Belief comes in various forms. There is *disbelief* (believing a statement to be false), *nonbelief* (not believing a statement to be true), *half-belief*, *quarter-belief*, and so on (the degrees of belief range from barely accepting a statement, to having complete conviction a statement is true). Knowledge could be defined as belief plus complete conviction and conclusive justification.

The following are two approaches as to how beliefs might be managed.

1. The *foundation approach* argues that beliefs are derived from reasons for these beliefs. A belief is justified if and only if (1) the belief is self-evident and (2) the belief can be derived from the set of other justified beliefs (circularity is not allowed).

2. The *coherence approach* argues that where beliefs originated is of no concern. Instead, beliefs must be logically coherent with other beliefs (believed by an individual). These beliefs can mutually justify each other and circularity is allowed. A number of different types of coherence have been proposed, Including *deductive coherence* (requires a logically consistent set of beliefs), *probabilistic coherence* (assigns probabilities to beliefs and applies the requirements of mathematical probability to them), *semantic coherence* (based on beliefs that have similar meanings), and *explanatory coherence* (requires that there be a consistent explanatory relationship between beliefs).

The *foundation approach* is very costly (in cognitive effort) to operate. For instance, the reasons for beliefs need to be remembered and applied when considering new beliefs. Studies[1181] show that people exhibit a belief preservation effect; they continue to hold beliefs after the original basis for those beliefs no longer

holds. The evidence suggests that people use some form of *coherence approach* for creating and maintaining their beliefs.

There are two different ways doubt about a fact can occur. When the truth of a statement is not known because of a lack of information, but the behavior in the long run is known, we have *uncertainty*. For instance, the outcome of the tossing of a coin is uncertain, but in the long run the result is known to be heads (or tails) 50% of the time. The case in which truth of a statement can never be precisely specified (indeterminacy of the average behavior) is known as *imprecision*; for instance, "it will be sunny tomorrow". It is possible for a statement to contain both uncertainty and imprecision. For instance, the statement, "It is likely that John is a young fellow", is uncertain (John may not be a *young fellow*) and imprecise (*young* does not specify an exact age). For a mathematical formulation, see Paskin.[1061]

Coding guidelines need to take into account that developers are unlikely to make wholesale modifications to their existing beliefs to make them consistent with any guidelines they are expected to adhere to. Learning about guidelines is a two-way process. What a developer already knows will influence how the guideline recommendations themselves will be processed, and the beliefs formed about their meaning. These beliefs will then be added to the developer's existing personal beliefs.[1472]

16.2.8.9.1 The Belief-Adjustment model

A belief may be based on a single piece of evidence, or it may be based on many pieces of evidence. How is an existing belief modified by the introduction of new evidence? The belief-adjustment model of Hogarth and Einhorn[581] offers an answer to this question. This subsection is based on that paper. The basic equation for this model is:

$$S_k = S_{k-1} + w_k[s(x_k) - R] \tag{0.23}$$

where: S_k is the degree of belief (a value between 0 and 1) in some hypothesis, impression, or attitude after evaluating k items of evidence; S_{k-1} is the anchor, or prior opinion ($S_0 i$ denotes the initial belief). $s(x_k)$ is the subjective evaluation of the kth item of evidence (different people may assign different values for the same evidence, x_k); R is the reference point, or background, against which the impact of the kth item of evidence is evaluated. w_k is the adjustment weight (a value between zero and one) for the kth item of evidence.

The encoding process

When presented with a statement, people can process the evidence it contains in several ways. They can use an *evaluation* process or an *estimation* process.

The *evaluation* process encodes new evidence relative to a fixed point— the hypothesis addressed by a belief. If the new evidence supports the hypothesis, a person's belief is increased, but that belief is decreased if it does not support the hypothesis. This increase, or decrease, occurs irrespective of the current state of a person's belief. For this case $R = 0$, and the belief-adjustment equation simplifies to:

$$S_k = S_{k-1} + w_k s(x_k) \tag{0.24}$$

where: $-1 \leq s(x_k) \leq 1$

An example of an evaluation process might be the belief that the object `X` always holds a value that is numerically greater than `Y`.

The *estimation* process encodes new evidence relative to the current state of a person's beliefs. For this case $R = S_{k-1}$, and the belief-adjustment equation simplifies to:

$$S_k = S_{k-1} + w_k(s(x_k) - S_{k-1}) \tag{0.25}$$

where: $0 \leq s(x_k) \leq 1$

In this case the degree of belief, in a hypothesis, can be thought of as a moving average. For an estimation process, the order in which evidence is presented can be significant. While reading source code written by somebody else, a developer will form an opinion of the quality of that person's work. The judgment of each code sequence will be based on the readers current opinion (at the time of reading) of the person who wrote it.

Processing

It is possible to consider $s(x_k)$ as representing either the impact of a single piece of evidence (so-called *Step-by-Step*, SbS), or the impact of several pieces of evidence (so-called *End-of-Sequence*, EoS).

$$S_k = S_0 + w_k[s(x_1, \ldots, x_k) - R] \tag{0.26}$$

where $s(x_1, \ldots, x_k)$ is some function, perhaps a weighted average, of the individual subjective evaluations.

If a person is required to give a Step-by-Step response when presented with a sequence of evidence, they obviously have to process the evidence in this mode. A person who only needs to give an End-of-Sequence response can process the evidence using either SbS or EoS. The process used is likely to depend on the nature of the problem. Aggregating, using EoS, evidence from a long sequence of items of evidence, or a sequence of complex evidence, is likely to require a large amount of cognitive processing, perhaps more than is available to an individual. Breaking a task down into smaller chunks by using an SbS process, enables it to be handled by a processor having a limited cognitive capacity. Hogarth and Einhorn proposed that when people are required to provide an EoS response they use an EoS process when the sequence of items is short and simple. As the sequence gets longer, or more complex, they shift to an SbS process, to keep the peak cognitive load (of processing the evidence) within their capabilities.

Adjustment weight

The adjustment weight, w_k, will depend on the sign of the impact of the evidence, $[s(x_k) - R]$, and the current level of belief, S_k. Hogarth and Einhorn argue that when $s(x_k) \leq R$:

$$w_k = \alpha S_{k-1} \tag{0.27}$$
$$S_k = S_{k-1} + \alpha S_{k-1} s(x_k) \tag{0.28}$$

and that when $s(x_k) > R$:

$$w_k = \beta(1 - S_{k-1}) \tag{0.29}$$
$$S_k = S_{k-1} + \beta(1 - S_{k-1}) s(x_k) \tag{0.30}$$

where α and β ($0 \leq \alpha, \beta \leq 1$) represent sensitivity toward positive and negative evidence. Small values indicating low sensitivity to new evidence and large values indicating high sensitivity. The values of α and β will also vary between people. For instance, some people have a tendency to give negative evidence greater weight than positive evidence. People having strong attachments to a particular point of view may not give evidence that contradicts this view any weight.[1339]

Order effects

It can be shown[581] that use of an SbS process when $R = S_{k-1}$ leads to a recency effect. When $R = 0$, a recency effect only occurs when there is a mixture of positive and negative evidence (there is no recency effect if the evidence is all positive or all negative).

The use of an EoS process leads to a primacy effect; however, a task may not require a response until all the evidence is seen. If the evidence is complex, or there is a lot of it, people may adopt an SbS process. In this case, the effect seen will match that of an SbS process.

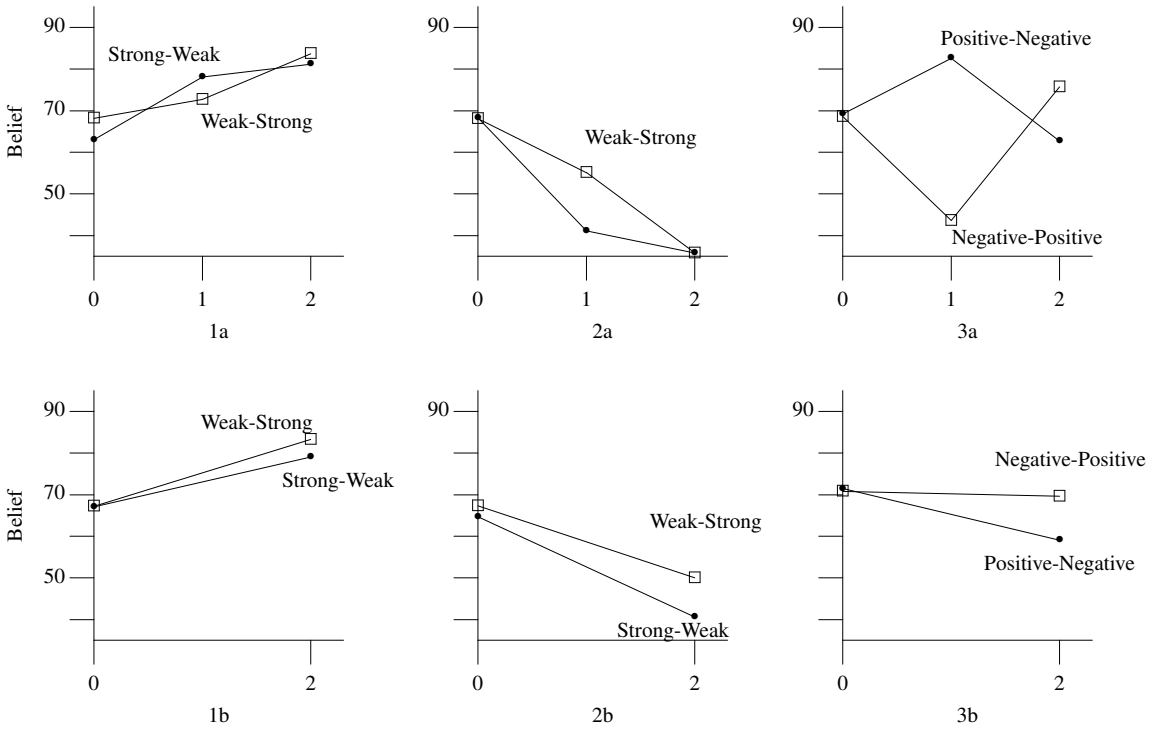


Figure 0.33: Subjects belief response curves for positive weak–strong, negative weak–strong, and positive–negative evidence; (a) Step-by-Step, (b) End-of-Sequence. Adapted from Hogarth.[581]

A *recency effect* occurs when the most recent evidence is given greater weight than earlier evidence. A *primacy effect* occurs when the initial evidence is given greater weight than later evidence.

recency effect
primacy effect

Study

A study by Hogarth and Einhorn[581] investigated order, and response mode, effects in belief updating. Subjects were presented with a variety of scenarios (e.g., a defective stereo speaker thought to have a bad connection, a baseball player whose hitting has improved dramatically after a new coaching program, an increase in sales of a supermarket product following an advertising campaign, the contracting of lung cancer by a worker in a chemical factory). Subjects read an initial description followed by two or more additional items of evidence. The additional evidence might be positive (e.g., "The other players on Sandy's team did not show an unusual increase in their batting average over the last five weeks") or negative (e.g., "The games in which Sandy showed his improvement were played against the last-place team in the league"). This positive and negative evidence was worded to create either strong or weak forms.

The evidence was presented in a variety of orders (positive or negative, weak or strong). Subjects were asked, "Now, how likely do you think X caused Y on a scale of 0 to 100?" In some cases, subjects had to respond after seeing each item of evidence: in other cases, subjects had to respond after seeing all the items.

The results (see Figure 0.33) only show a recency effect when the evidence is mixed, as predicted for the case $R = 0$.

Other studies have duplicated these results. For instance, professional auditors have been shown to display recency effects in their evaluation of the veracity of company accounts.[1062, 1368]

16.2.8.9.2 Effects of beliefs

The persistence of beliefs after the information they are based on has been discredited is an important issue in developer training.

Studies of physics undergraduates[911] found that many hours of teaching only had a small effect on their

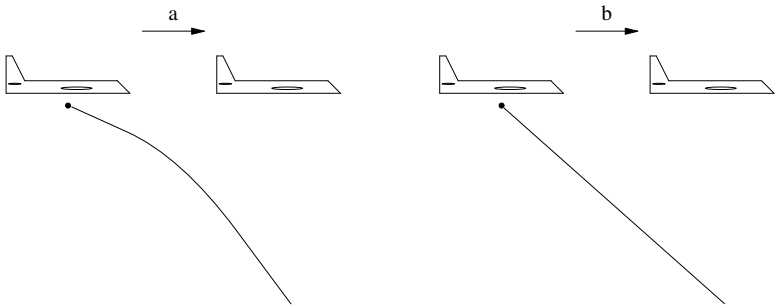


Figure 0.34: Two proposed trajectories of a ball dropped from a moving airplane. Based on McCloskey.[911]

qualitative understanding of the concepts taught. For instance, predicting the motion of a ball dropped from an airplane (see Figure 0.34). Many students predicted that the ball would take the path shown on the right (*b*). They failed to apply what they had been taught over the years to pick the path on the left (*a*).

A study by Ploetzner and VanLehn[1095] investigated subjects who were able to correctly answer these conceptual problems. They found that the students were able to learn and apply information that was implicit in the material taught. Ploetzner and VanLehn also built a knowledge base of 39 rules needed to solve the presented problems, and 85 rules needed to generate the incorrect answers seen in an earlier study.

A study by Pazzani[1066] showed how beliefs can increase, or decrease, the amount of effort needed to deduce a concept. Two groups of subjects were shown pictures of people doing something with a balloon. The balloons varied in color (yellow or purple) and size (small or large), and the people (adults or five-year-old children) were performing some operation (stretching balloons or dipping them in water). The first group of subjects had to predict whether the picture was an "example of an *alpha*", while the second group had to "predict whether the balloon will be inflated". The picture was then turned over and subjects saw the answer. The set of pictures was the same for both groups of subjects.

The conditions under which the picture was an *alpha* or *inflate* were the same, a conjunctive condition `(age == adult) || (action == stretching)` and a disjunction condition `(size == small) && (color == yellow)`.

The difference between these two tasks to predict is that the first group had no prior beliefs about *alpha* situations, while it was assumed the second group had background knowledge on inflating balloons. For instance, balloons are more likely to inflate after they have been stretched, or an adult is doing the blowing rather than a child.

conditionals 1256
conjunctive/disjunctive

The other important point to note is that people usually require more effort to learn conjunctive conditions than they do to learn disjunctive conditions.

The results (see Figure 0.35) show that, for the *inflate* concept, subjects were able to make use of their existing beliefs to improve performance on the disjunctive condition, but these beliefs caused a decrease in performance on the conjunctive condition (being small and yellow is not associated with balloons being difficult to inflate).

A study by Gilbert, Tafarodi, and Malone[490] investigated whether people could comprehend an assertion without first believing it. The results suggested that their subjects always believed an assertion presented to them, and that only once they had comprehended it were they in a position to, possibly, *unbelieve* it. The experimental setup used, involved presenting subjects with an assertion and interrupting them before they had time to *unbelieve* it. This finding has implications for program comprehension in that developers sometimes only glance at code. Ensuring that what they see does not subsequently need to be *unbelieved*, or is a partial statement that will be read the wrong way without other information being provided, can help prevent people from acquiring incorrect beliefs. The commonly heard teaching maxim of "always use correct examples, not incorrect ones" is an application of this finding.

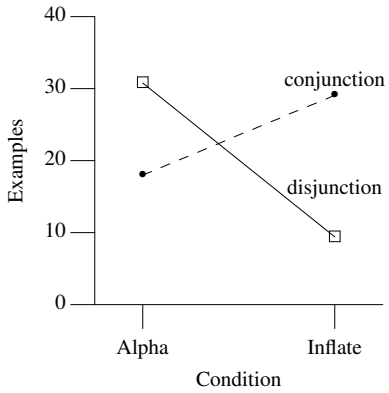


Figure 0.35: Number of examples needed before *alpha* or *inflate* condition correctly predicted in six successive pictures. Adapted from Pazzani[1066]

16.2.8.10 Confirmation bias

There are two slightly different definitions of the term *confirmation bias* used by psychologists, they are: confirmation bias

1. A person exhibits confirmation bias if they tend to interpret ambiguous evidence as (incorrectly) confirming their current beliefs about the world. For instance, developers interpreting program behavior as supporting their theory of how it operates, or using the faults exhibited by a program to conform their view that it was poorly written.
2. When asked to discover a rule that underlines some pattern (e.g., the numeric sequence 2–4–6), people nearly always apply test cases that will confirm their hypothesis. They rarely apply test cases that will falsify their hypothesis.

Rabin and Schrag[1134] built a model showing that confirmation bias leads to overconfidence (people believing in some statement, on average, more strongly than they should). Their model assumes that when a person receives evidence that is counter to their current belief, there is a positive probability that the evidence is misinterpreted as supporting this belief. They also assume that people always correctly recognize evidence that confirms their current belief. Compared to the correct statistical method, Bayesian updating, this behavior is biased toward confirming the initial belief. Rabin and Schrag showed that, in some cases, even an infinite amount of evidence would not necessarily overcome the effects of confirmatory bias; over time a person may conclude, with near certainty, that an incorrect belief is true. 0 overconfidence

The second usage of the term *confirmation bias* applies to a study performed by Wason,[1444] which became known as the *2–4–6 Task*. In this study subjects were asked to discover a rule known to the experimenter. They were given the initial hint that the sequence 2–4–6 was an instance of this rule. Subjects had to write down sequences of numbers and show them to the experimenter who would state whether they did, or did not, conform to the rule. When they believed they knew what the rule was, subjects had to write it down and declare it to the experimenter. For instance, if they wrote down the sequences 6–8–10 and 3–5–7, and were told that these conformed to the rule, they might declare that the rule was *numbers increasing by two*. However, this was not the experimenters rule, and they had to continue generating sequences. Wason found that subjects tended to generate test cases that confirmed their hypothesis of what the rule was. Few subjects generated test cases in an attempt to disconfirm the hypothesis they had. Several subjects had a tendency to declare rules that were mathematically equivalent variations on rules they had already declared.

8 10 12: two added each time; **14 16 18:** even numbers in order of magnitude; **20 22 24:** same reason; **1 3 5:** two added to preceding number.
The rule is that by starting with any number two is added each

time to form the next number.
2 6 10: middle number is the arithmetic mean of the other two;
1 50 99: same reason.
The rule is that the middle number is the arithmetic mean of the other two.
3 10 17: same number, seven, added each time; **0 3 6**;
three added each time.
The rule is that the difference between two numbers next to each other is the same.
12 8 4: the same number subtracted each time to form the next number.
The rule is adding a number, always the same one to form the next number.
1 4 9: any three numbers in order of magnitude.
The rule is any three numbers in order of magnitude.
Sample 2-4-6 subject protocol. Adapted from Wason.[1444]

The actual rule used by the experimenter was "three numbers in increasing order of magnitude".

These findings have been duplicated in other studies. In a study by Mynatt, Doherty, and Tweney,[989] subjects were divided into three groups. The subjects in one group were instructed to use a confirmatory strategy, another group to use a disconfirmatory strategy, and a control group was not told to use any strategy. Subjects had to deduce the physical characteristics of a system, composed of circles and triangles, by firing particles at it (the particles, circles and triangles, appeared on a computer screen). The subjects were initially told that "triangles deflect particles". In 71% of cases subjects selected confirmation strategies. The instructions on which strategy to use did not have any significant effect.

In a critique of the interpretation commonly given for the results from the 2–4–6 Task, Klayman and Ha[745] pointed out that it had a particular characteristic. The hypothesis that subjects commonly generate (*numbers increasing by two*) from the initial hint is completely contained within the experimenters rule, case 2 in Figure 0.36. Had the experimenters rule been *even numbers increasing by two*, the situation would have been that of case 3 in Figure 0.36.

Given the five possible relationships between hypothesis and rule, Klayman and Hu analyzed the possible strategies in an attempt to find one that was optimal for all cases. They found that the optimal strategy was a function of a variety of task variables, such as the base rates of the target phenomenon and the hypothesized conditions. They also proposed that people do not exhibit confirmation bias, rather people have a general all-purpose heuristic, the *positive test strategy*, which is applied across a broad range of hypothesis-testing tasks.

A *positive test strategy* tests a hypothesis by examining instances in which the property or event is expected to occur to see if it does occur. The analysis by Klayman and Hu showed that this strategy performs well in real-world problems. When the target phenomenon is relatively rare, it is better to test where it occurs (or where it was known to occur in the past) rather than where it is not likely to occur.

A study by Mynatt, Doherty, and Dragan[988] suggested that capacity limitations of working memory were also an issue. Subjects did not have the capacity to hold information on more than two alternatives in working memory at the same time. The results of their study also highlighted the fact that subjects process the alternatives in *action* (what to do) problems differently than in *inference* (what is) problems.

Karl Popper[1104] pointed out that scientific theories could never be shown to be logically true by generalizing from confirming instances. It was the job of scientists to try to perform experiments that attempted to falsify a theory. Popper's work on how a hypothesis should be validated has become the generally accepted way of measuring performance (even if many scientists don't appear to use this approach).

The fact that people don't follow the hypothesis-testing strategy recommended by Popper is seen, by some, as a deficiency in peoples thinking processes. The theoretical work by Klayman and Hu shows that it might

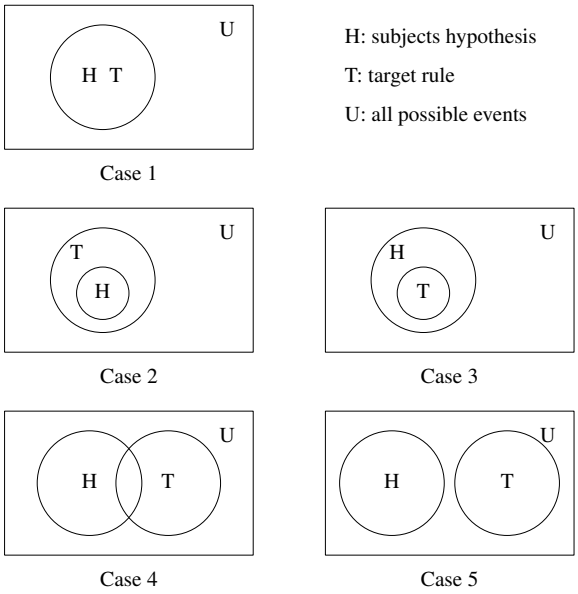


Figure 0.36: Possible relationships between hypothesis and rule. Adapted from Klayman.[745]

be Poppers theories that are deficient. There is also empirical evidence showing that using disconfirmation does not necessarily improve performance on a deduction task. A study by Tweney, Doherty, Worner, Pliske, Mynatt, Gross, and Arkkelin[1378] showed that subjects could be trained to use a disconfirmation strategy when solving the 2–4–6 Task. However, the results showed that using this approach did not improve performance (over those subjects using a confirmation strategy).

Do developers show a preference toward using positive test strategies during program comprehension? What test strategy is the best approach during program comprehension? The only experimental work that has addressed this issue used students in various stages of their academic study. A study by Teasley, Leventhal, Mynatt, and Rohlman[1334] asked student subjects to test a program (based on its specification). The results showed that the more experienced subjects created a greater number of negative tests.

16.2.8.11 Age-related reasoning ability

reasoning ability age-related

It might be thought that reasoning ability declines with age, along with the other faculties. A study by Tentori, Osherson, Hasher, and May[1336] showed the opposite effect; some kinds of reasoning ability improving with age.

Consider the case of a person who has to decide between two alternatives, A and B (e.g., vanilla and strawberry ice cream), and chooses A. Adding a third alternative, C (e.g., chocolate ice cream) might entice that person to select C. A mathematical analysis shows that adding alternative C would not cause a change of preference to B. How could adding the alternative chocolate ice cream possibly cause a person who previously selected vanilla to now choose strawberry?

So-called *irregular choices* have been demonstrated in several studies. Such irregular choices seem to occur among younger (18–25) subjects, older (60–75) subjects tending to be uninfluenced by the addition of a third alternative.

16.3 Personality

developer personality

To what extent does personality affect developers' performance, and do any personality differences need to be reflected in coding guidelines?

- A study by Turley and Bieman[1370] looked for differences in the competencies of exceptional and non-exceptional developers. They found the personal attributes that differentiated performances were:

desire to contribute, perseverance, maintenance of a *big picture* view, desire to do/bias for action, driven by a sense of mission, exhibition and articulation of strong convictions, and proactive role with management. Interesting findings in another context, but of no obvious relevance to these coding guidelines. Turley and Bieman also performed a Myers-Briggs Type Indicator (MBTI) test[987] on their subjects. The classification containing the most developers (7 out of 20) was INTJ (*Introvert*, *Intuitive*, *Thinking*, *Judging*), a type that occurs in only 10% of male college graduates. In 15 out of 20 cases, the type included *Introvert, Thinking*. There was no significance in scores between exceptional and nonexceptional performers. These findings are too broad to be of any obvious relevance to coding guidelines.

Myers-Briggs Type Indicator

- A study of the relationship between personality traits and achievement in an introductory Fortran course was made by Kagan and Douthat.[699] They found that relatively introverted students, who were hard-driving and ambitious, obtained higher grades then their more extroverted, easy-going compatriots. This difference became more pronounced as the course progressed and became more difficult. Again these findings are too broad to be of any obvious relevance to these coding guidelines.

These personality findings do not mean that to be a good developer a person has to fall within these categories, only that many of those tested did.

It might be assumed that personality could affect whether a person enjoys doing software development, and that somebody who enjoys their work is likely to do a better job (but does personal enjoyment affect quality, or quantity of work performed?). These issues are considered to be outside the scope of this book (they are discussed a little more in Staffing,).

coding 0 guidelines staffing

Developers are sometimes said to be paranoid. One study[1469] has failed to find any evidence for this claim.

Usage

17 Introduction

Usage 1 Usage introduction

This subsection provides some background on the information appearing in the Usage subsections of this book. The purpose of this usage information is two-fold:

1. To give readers a feel for the common developer usage of C language constructs. Part of the process of becoming an experienced developers involves learning about what is common and what is uncommon. However, individual experiences can be specific to one application domain, or company cultures.
2. To provide frequency-of-occurrence information that could be used as one of the inputs to cost/benefit decisions (i.e., should a guideline recommendation be made rather than what recommendation might be made). This is something of a chicken-and-egg situation in that knowing what measurements to make requires having potential guideline recommendations in mind, and the results of measurements may suggest guideline recommendations (i.e., some construct occurs frequently).

guideline 0 recommen-dations selecting

Almost all published measurements on C usage are an adjunct to a discussion of some translator optimization technique. They are intended to show that the optimization, which is the subject of the paper, is worthwhile because some constructs occurs sufficiently often for an optimization to make worthwhile savings, or that some special cases can be ignored because they rarely occur. These kinds of measurements are usually discussed in the *Common implementation* subsections. One common difference between the measurements in Common Implementation subsections and those in Usage subsections is that the former are often dynamic (instruction counts from executing programs), while the latter are often static (counts based on some representation of the source code).

There have been a few studies whose aim has been to provide a picture of the kinds of C constructs that commonly occur (e.g., preprocessor usage,[396] embedded systems[390]). These studies are quoted in the

relevant C sentences. There have also been a number of studies of source code usage for other algorithmic languages, Assembler,[292] Fortran,[750] PL/1,[387] Cobol[12, 226, 662] (measurements involving nonalgorithmic languages have very different interests[208, 240]). These are of interest in studying cross-language usage, but they are not discussed in this book. In some cases a small number of machine code instruction sequences (which might be called idioms) have been found to account for a significant percentage of the instructions executed during program execution.[1269]

The intent here is to provide a broad brush picture. On the whole, single numbers are given for the number of occurrences of a construct. In most cases there is no break down by percentage of functions, source files, programs, application domain, or developer. There is variation across all of these (e.g., application domain and individual developer). Whenever this variation might be significant, additional information is given. Those interested in more detailed information might like to make their own measurements.

0 coding guidelines applications
0 coding guidelines coding style

Many of the coding guideline recommendations made in this book apply to the visible source code as seen by the developer. For these cases any usage measurements also apply to the visible source code. The effects of any macro replacement, conditional inclusion, or **#include**d header are ignored. Each usage subsection specifies what the quoted numbers apply to (usually either visible source, or the tokens processed during translation phase 7).

macro replacement

In practice many applications do not execute in isolation; there is usually some form of operating system that is running concurrently with it. The design of processor instruction sets often takes task-switching and other program execution management tasks into account. In practice the dynamic profile of instructions executed by a processor reflects this mix of usage,[119] as does the contents of its cache.[906]

17.1 Characteristics of the source code

All source code may appear to look the same to the casual observer. An experienced developer will be aware of recurring patterns; source can be said to have a style. Several influences can affect the characteristics of source code, including the following:

source code characteristics
0 coding guidelines coding style

- *Use of extensions to the C language and differences, for prestandard C, from the standard (often known as K&R C).* Some extensions eventually may be incorporated into a revised version of the standard; for instance, **long long** was added in C99. Some extensions are specific to the processor on which the translated program is to execute.

0 common implementations language specification

- *The application domain.* For instance, scientific and engineering applications tend to make extensive use of arrays and spend a large amount of their time in loops processing information held in these arrays; screen based interactive applications often contain many calls to GUI library functions and can spend more time in these functions than the developer's code; data-mining applications can spend a significant amount of time searching large data structures.

- *How the application is structured.* Some applications consist of a single, monolithic, program, while others are built from a collection of smaller programs sharing data with one another. These kinds of organization affect how types and objects are defined and used.

- *The extent to which the source has evolved over time.* Developers often adopt the low-risk strategy of making the minimal number of changes to a program when modifying it. Often this means that functions and sequences of related statements tend to grow much larger than would be the case if they had been written from scratch, because no restructuring is performed.

0 application evolution

- *Individual or development group stylistic usage.* These differences can include the use of large or small functions, the use of enumeration constants or object-like macros, the use of the smallest integer type required rather than always using **int**, and so forth.

1931 macro object-like

17.2 What source code to measure?

This book is aimed at a particular audience and the source code they are likely to be actively working on. This audience will be working on C source that has been written by more than one developer, has existed for a year or more, and is expected to continue to be worked on over the coming years.

benchmarks 0

The benchmarks used in various application areas were written with design aims that differ from those of this book. For instance, the design aim behind the choice of programs in the SPEC CPU benchmark suite was to measure processor, memory hierarchy, and translator performance. Many of these programs were written by individuals, are relatively short, and have not changed much over time.

Although there is a plentiful supply of C source code publicly available (an estimated 20.3 million C source files on the Web[122]), this source is nonrepresentative in a number of ways, including:

- The source has had many of the original defects removed from it. The ideal time to make these measurements is while the source is being actively developed.
- Software for embedded systems is often so specialized (in the sense of being tied to custom hardware), or commercially valuable, that significant amounts of it are not usually made publicly available.

Nevertheless, a collection of programs was selected for measurement, and the results are included in this book (see Table 0.23). The programs used for this set of measurements have reached the stage that somebody has decided that they are worth releasing. This means that some defects in the source, prior to the release, will not be available to be included in these usage figures.

Table 0.23: Programs whose source code (i.e., the `.c` and `.h` files) was used as the input to measurement tools (operating on either the visible or translated forms), whose output was used to generate this book's usage figures and tables.

Name	Application Domain	Version
gcc	C compiler	2.95
idsoftware	Games programs, e.g., Doom	
linux	Operating system	2.4.20
mozilla	Web browser	1.0
openafs	File system	1.2.2a
openMotif	Window manager	2.2.2
postgresql	Database system	6.5.3

Table 0.24: Source files excluded from the Usage measurements.

Files	Reason for Exclusion
gcc-2.95/libio/tests/tfformat.c	a list of approximately 4,000 floating constants
gcc-2.95/libio/tests/tiformat.c	a list of approximately 5,000 hexadecimal constants

Table 0.25: Character sequences used to denote those operators and punctuators that perform more than one role in the syntax.

Symbol	Meaning	Symbol	Meaning
`++v`	prefix ++	`--v`	prefix --
`v++`	postfix ++	`v--`	postfix --
`-v`	unary minus	`+v`	unary plus
`*v`	indirection operator	`*p`	star in pointer declaration
`&v`	address-of		
`:b`	colon in bitfield declaration	`?:`	colon in ternary operator

17.3 How were the measurements made?

The measurements were based two possible interpretations of the source (both of them static, that is, based on the source code, not program execution):

- *The visible source.* This is the source as it might be viewed in a source code editor. The quoted results specify whether the `.c` or the `.h` files, or both, were used. The tools used to make these measurements are based on either analyzing sequences of characters or sequences of preprocessing tokens (built from the sequences of characters). The source of the tools used to make these measurements is available on this book's Web site: http://www.knosof.co.uk/cbook/cbook.html.
- *The translated source.* This is the source as processed by a translator following the syntax and semantics of the C language. Measurements based on the translated source differ from those based on the visible source in that they may not include source occurring within some arms of conditional inclusion directives, may be affected by macro replacement, may not include all source files in the distribution (because the make-file does not require them to be translated), and do not include a few files which could not be successfully translated by the tool used. (The tools used to measure the translated source were based on a C static analysis tool.[681]) Every attempt was made to exclude the contents of any **#include**d system headers (i.e., any header using the < > delimited form) from the measurements. However, the host on which the measurements were made (RedHat 9, a Linux distribution) will have some effect; for instance, use of a macro defined in an implementation's header may expand to a variety of different preprocessing tokens, depending on the implementation. Also some application code contains conditional inclusion directives that check properties of the host O/S.

conditional inclusion macro replacement

Note. The condition for inclusion in a table containing *Common token pairs involving* information was that percentage occurrence of both tokens be greater than 1% and that the sum of both token frequencies be greater than 5%. In some cases the second requirement excluded tokens pairs when the percentage occurrence of one of the tokens was relatively high. For instance, the token pair `- character-constant` does not appear in Table 866.3 because the sum of the token frequencies is 4.1 (i.e., 1.9+2.2).

The usage information often included constructs that rarely occurred. Unless stated otherwise a cut-off of 1% was used. Values for table entries such as *other-types* were created by summing the usage information below this cut-off value.

1. Scope

1 This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.[1)]

standard specifies form and interpretation

Commentary

The C Standard describes the behavior of programs (not always in complete detail, an implementation is given various amounts of leeway in translating some constructs). The behavior of implementations has to be deduced from the need to implement the described behavior of programs.

The committee took the view that programs are more important than implementations. This principle was and is used during the decision-making process of the C Standard Committee. Implementors sometimes argued that what their implementation did was/is important. The particular characteristics of an implementation can influence the usage of C language in programs, as can the characteristics of the host (e.g., the width of integer types supported). The Committee preferred to consider the extent of usage in existing programs and only became involved in the characteristics of implementations when there was widespread usage of a particular construct.

Existing code is important, existing implementations are not. Rationale

C++

1.1p1 *This International Standard specifies requirements for implementations of the C++ programming language.*

The C++ Standard does not specify the behavior of programs, but of implementations. For this standard the behavior of C++ programs has to be deduced from this, implementation-oriented, specification.

In those cases where the same wording is used in both standards, there is the potential for a different interpretation. In the case of the preprocessor, an entire clause has been copied, almost verbatim, from one document into the other. Given the problems that implementors are having producing a translator that handles the complete C++ Standard, and the pressures of market forces, it might be some time before people become interested in these distinctions.

Other Languages

This exact wording appears in both the Cobol and Fortran standards (except the language name is changed and Fortran programs are "expressed" rather than "written"). Some language definitions do not explicitly specify whether they apply to programs or implementations. Pascal defines conformance requirements for both implementations and programs.

Gosling[508] *We intend that the behavior of every language construct is specified here, so that all implementations of Java will accept the same programs.*

Common Implementations

base document The C language was first described in 1975 in a Bell Labs technical report[1170] (the successor to a language called B[674]). The more commonly known book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie[725] was published in 1978. There was also a report published in the same year listing recent changes.[726] A second edition of this book was published after the ANSI C Standard was ratified[727] which updated its description of the language to follow that given in the newly published standard. There has been no republication since the C99 revision of the Standard.

The first edition of the Kernighan and Ritchie book describes what became known as *K&R C*. A large number of implementations were based on the original K&R book, few of them adhering exactly to the specification it contained, (because it was open to interpretation). The term K&R compiler is often applied generically to translators that do not support function prototypes (an easily spotted characteristic).

As time has passed the number of implementations, in use, based on K&R C has dropped dramatically. But there is still source code in use that was written to the K&R specification. Vendors like to keep their customers happy by translating their existing code and many have added *support K&R* options to their products.

The base document for the library clause was the *1984 /usr/group Standard* published by the /usr/group Standard's Committee, Santa Clara, California, USA.

Coding Guidelines

Coding guidelines invariably specify that ISO 9899 is the definitive document. The problem at the time of writing, and for the next few years, is that most implementations in common use follow the 1990 document, not the 1999 revision. Most of the changes involve additional functionality with very few changes to existing behavior. So most of the updates that are needed to move to C99 can be handled as new material.

Up until the mid 1990s portability considerations meant having to keep an eye on maintaining a K&R compatibility option. These days platforms that only support K&R are limited to a few niches, where there is insufficient market interest to make it worthwhile to create an ISO-conforming implementation.

Many students who are taught C++ are told that it is a superset of C. This was not always the case in C90 and is not true in C99 (where there is additional support for functionality not available in C++). Some

compiler vendors offer a *C compiler* switch on their C++ compiler. Such switches do not always have the effect of creating a conforming C compiler. The issues of C/C++ compatibility are dealt with in the C++ subsections for each sentence.

---

2 It specifies

Commentary

This list is not exhaustive in that permission is explicitly given for an implementation to have extensions.

95 implementation extensions

C++

The C++ Standard does not list the items considered to be within its scope.

Coding Guidelines

These coding guideline subsections sometimes specify recommendations to be followed by developers for the usage of C language constructs.

---

3 — the representation of C programs;

program specify representation

Commentary

The representation described is essentially the same as written text, with special meaning attributed to certain characters, singly or in sequences (e.g., end-of-line indicators). The representation also includes how the components of a C program are organized. In most cases files are used.

116 translation phase 1

108 source files

The representation of C programs occurs at many different levels. There is the representation as it appears to the developer, the bytes read from media by the operating system, and the pattern of bits held on storage media. The representation we are interested in is the one that appears, to a developer, when viewed with an editor that supports the characters required by the C Standard.

Common Implementations

The C language was first implemented on hosts that used the Ascii character set. The EBCDIC (Extended Binary-Coded-Decimal Interchange Code) character set is commonly used on mainframes and C can be represented using this character set.

EBCDIC

Use of C for embedded applications and the prominence of Japanese in this area meant that C was the first standardized language to allow the writing of programs that contained other character sets.

---

4 — the syntax and constraints of the C language;

Commentary

These two sets of specifications are set in concrete and must be implemented, as written, by every conforming implementation. These specifications appear in the Standard within clauses headed by "Syntax", or by "Constraints".

81 conformance

63 constraint

C++

The first such requirement is that they implement the language, and so this International Standard also defines C++.

1.1p1

While the specification of the C++ Standard includes syntax, it does not define and use the term *constraints*. What the C++ specification contains are *diagnosable rules*. A conforming implementation is required to check and issue a diagnostic if violated.

146 diagnostic shall produce

Common Implementations

Some implementations add additional syntax. Adding additional constraints, or relaxing the existing ones is not commonly seen in implementations, but it does occur.

Coding Guidelines

Constructs that violate C syntax or constraints are required to be diagnosed by a conforming implementation. Working programs rarely contain such constructs (unless there is a bug in the implementation; for instance, `gcc` allows semicolons to be omitted in several places). Duplicating these requirements in coding guidelines does not add value.

Implementations that provide extensions to standard C do not always fully define these extensions. In particular they often define how an extension may be used but fail to define what the constraints on its use are. The coding guideline issues relating to use of extensions is discussed elsewhere.

extensions 95.1
cost/benefit

— the semantic rules for interpreting C programs; 5

Commentary

These rules appear under the clause heading *Semantics*, or sometimes *Description*, along with definitions of conformance. The behavior arising out of these semantic rules is what developers use to write programs that have an external effect.

conformance 81
behavior 41

The standard talks about an abstract machine and how programs are to be interpreted as-if running under it. Unfortunately, the specifications given in the standard are not worded in a form that directly addresses the properties of this machine; as such, this machine is never fully defined. However, the semantic rules specified in the C Standard are not all set in stone. An implementation may be required to select among several alternatives (these form the category of unspecified behaviors), chose its own behavior (these form the category of implementation-defined behaviors), or the standard may not impose any requirements on the behavior (these form the category of undefined behaviors).

C++

The C++ Standard specifies rules for implementations, not programs.

Coding Guidelines

A good first approximation to a set of coding guidelines is to recommend against the use of constructs whose semantic rules can vary across implementations (Annex J summarizes these and the majority of the rules in the MISRA C guidelines are based on this principle). While many of these Coding guideline subsections discuss the effect of implementation differences, these are only treated as a possible contributing factor to the primary consideration (i.e., cost) and not as a rationale in their own right.

MISRA 0
guideline 0
recommen-
dations
selecting

— the representation of input data to be processed by C programs; 6

Commentary

The C language had its beginnings in solving practical problems. Ignoring the representational issues of data was not a viable option. The Committee did adopt a specification for a set of I/O concepts (e.g., streams, binary and text files) and functions for manipulating them into the C library (the base document provided the underlying models).

base doc-1
ument

The underlying unit of input is the byte. The standard does not require that any sequence of bits within any byte have a particular interpretation (the functions provided by the **<ctype.h>** header can be applied to them, just like any other numeric quantity).

Other Languages

Some language standards committees have taken the view that I/O was not an important aspect of the language and provided a minimal set of functionality in this area, saying nothing about representational issues. In other languages, for instance Cobol, I/O is a significant part of that languages' specification.

C++

The C++ Standard is silent on this issue.

Common Implementations

A form of input not explicitly dealt with in the standard is the reading/writing data from/to registers, or I/O ports. This is a common form of I/O in freestanding environments.

Many implementations that support such functionality use the **volatile** type qualifier or some extension that allows objects to be placed at known locations in storage; reading values from such objects cause input to take place. Here the representation of the input value is interpreted according to the type of object through which it is accessed.

1476 type qualifier syntax

Coding Guidelines

The issues involved in converting this input data into some internal form is an application domain issue that is outside the scope of these coding guidelines. For instance, a floating-point number presented as a sequence of characters, on an input stream, may contain more accuracy than can be represented by the host. There is a guideline recommendation dealing with the use of representation information.

569.1 representation information using

7— the representation of output data produced by C programs;

Commentary

The standard does not specify any representation in terms of bit patterns, or pixels on a display device. However, the standard does specify some ordering requirements on output data. Data written out can be read back in to produce the same value. Output written to a display device will appear in the order it is written (but nothing is said about left-to-right, right-to-left, top-to-bottom, or any other visible ordering on the device).

254 writing direction locale-specific

C supports two forms of representation on output, text and binary. Text I/O is structured into lines of characters (there can be a great deal of variability in the external representation of characters written to text streams), while binary I/O is an ordered sequence of bytes.

Other Languages

In some application domains organizing the output data is a substantial part of the problem. Some languages, for instance Cobol, have mechanisms that provide application domain related control (in the case of Cobol the formatting of numeric quantities) of the output produced by a program.

Common Implementations

In most cases, implementations use the same representation for output data as they use for input data.

Coding Guidelines

The coding guideline subsections only discuss the representation of output data to the extent that it may be used as input data to other programs written in C.

8— the restrictions and limits imposed by a conforming implementation of C.

limits specify

Commentary

The Committee recognized that all implementations impose some limits on the size of programs that can be translated. They decided to face up to this issue by specifying minimum requirements. By specifying a list of limits, the Committee is attempting to guarantee a minimal level of support, for programs, by all conforming implementations. The limits were seen as a floor that implementations should strive to exceed, not as a ceiling they could stop at.

273 environmental limits
276 translation limits

Since the C90 Standard was written, the average amount of memory available to translators, on the majority of hosts, has increased significantly. For C99 the nominal translator host memory limit was increased to 512 K.

C90

The model of the minimal host expected to be able to translate a C program was assumed to have 64 K of free memory.

C++

Annex B contains an informative list of implementation limits. However, the C++ Standard does not specify any minimum limits that a conforming implementation must meet.

Common Implementations

Few implementations document all the limits they impose. This is usually because of the use of dynamic data structures, which means that their only fixed limit is the amount of memory available during translation.

This International Standard does not specify 9

Commentary

The C committee is up front about what the C Standard is not about.

C++

The C++ Standard does not list any issues considered to be outside of its scope.

Other Languages

Some languages standards include a list of items not specified by their respective documents, while others do not. Many of the items listed in the C Standard also appear in the Fortran Standard.

Coding Guidelines

coding guidelines background to 0

A set of coding guideline recommendations cannot hope to cover every issue that occurs in source code. Delimiting the areas not covered by a set of guidelines is as important as specifying those areas covered.

---

program transformation mechanism

— the mechanism by which C programs are transformed for use by a data-processing system; 10

Commentary

The standard uses the term *translator* to disassociate itself from known implementation techniques for transforming programs, such as compilers and interpreters. There is no requirement that the transformation process use programs that have been written in C, although the library does contain many of the support functions needed in the implementation of such a translator.

All suggestions requiring some mechanism to exist for passing options to a translator, at translation-time, were turned down by the Committee.

Rationale

One proposal long entertained by the C89 Committee was to mandate that each implementation have a translation-time switch for turning off extensions and making a pure Standard-conforming implementation. It was pointed out, however, that virtually every translation-time switch setting effectively creates a different "implementation", however close may be the effect of translating with two different switch settings. Whether an implementor chooses to offer a family of conforming implementations, or to offer an assortment of non-conforming implementations along with one that conforms, was not the business of the C89 Committee to mandate. The Standard therefore confines itself to describing conformance, and merely suggests areas where extensions will not compromise conformance.

implementation single pass

Many existing translators operate in a single pass and continued support for this form of implementation a consideration for many members of WG14 when considering the specification of C syntax and semantics.

Other Languages

Some standards use the term *language processor* to define what C calls a translator.

Common Implementations

The most common transformation mechanism (implementation technique) is to compile the program's source code to some form of machine code. This machine code could represent the instructions of a real processor (in the sense of being able to hold it in one's hand), or some virtual machine whose operations are performed in software (usually called an interpreter). In a few cases processors have been designed to execute a representation of some language directly. The Bell Labs CRISP processor[362] was designed to

efficiently support the execution of translated C programs (although the extent to which its instruction set might be claimed to be *C-like* is open to debate); the Symbolics 3600 processor used Lisp as its machine language;[430] the Novix NC4016[762] used Forth as its instruction set.

The ability to execute source code on a line-by-line basis is rarely provided (the Extensible Interactive C system[156] is an exception). In such an approach the standard still requires (if a vendor wanted to claim that their product was a conforming implementation) that the entire program's source code, even the unexecuted portions, be analyzed for syntax and constraint violations. As well as providing an interactive mode, the Extensible Interactive C system also provides a batch mode.

An advantage of the virtual machine approach is that the generated code can be executed unchanged on a wide variety of different processors, given the availability of a software interpreter. This is how Java achieves its portability. Another advantage of this approach is the compactness of the generated code. In applications where code size is more important than performance, it can be the deciding factor in choosing an interpretive approach.

The performance advantage obtained from using a cache shows how the execution time characteristics of many applications is to repeatedly execute the same sequences of instructions within short time periods. It is possible to make use of this execution time characteristic to have the best of both worlds— execution performance and compact code. The less-frequently executed portions of a program exist in virtual machine code form and the frequently executed portions in host processor machine code. For a VLIW processor, Hoogerbrugge[589] was able to obtain a 50% reduction in code size for a negligible increase in execution time.

0 cache

Whatever the mechanism used to transform C programs, it will invariably support some form of `-I` and `-D` options. These two options are almost universally used by C translators for specifying search paths for **`#include`** files and for defining macros (on the command line), respectively.

1898 #include places to search for
1931 macro object-like

Some translators access environment variables, of their host operating, to obtain the values of attribute that vary between different hosts. For instance, search paths for the **`#include`** directive, or the number of processors available to a program (when generating parallelization code[1314]).

1898 #include places to search for

Mixed forms of translation are being researched. Here translation of selected portions of a program occurs during the execution of that program. The advantage of this dynamic compilation approach is that it is possible to make use of runtime information to specialize the code, enhancing performance (provided the specialized execution savings are greater than the overhead of a dynamic compilation). The DyC toolset[513,950] has achieved some interesting results.

Many so-called *number crunching* applications are written in Fortran. A variety of parallel and vector processors have been built to reduce the execution time of such programs, which has entailed producing translators capable of vectorizing and parallelizing Fortran. One way to tap into this existing technology is to translate C source to Fortran.[722]

Downloading programs onto mobile devices, where they might only be executed once, is becoming more common. In this environment, the consumption of electrical power is an important consideration. A study by Palm and Moss[1044] performed a cost/benefit analysis of translating code on the client or server, with or without optimization. The energy quantities considered were: $E_{download}$, energy consumed by the wireless card while downloading code; $E_{wait-download}$, energy consumed by the client while waiting for code to download; $E_{wait-compile}$, energy consumed while waiting for code to compile or optimize on the server; $E_{compile}$, energy for compiling or optimizing on the client; and E_{run}, energy for running the compiled application on the client.

Coding Guidelines

Providing different options to a translator effectively creates different translators. The purpose of specifying options is to change the behavior of a translator, otherwise there is no point in specifying it. Whether use of an option radically changes the behavior of a translator (e.g., by enabling language extensions, changing the alignment of objects in memory, or selecting different hosts as the execution environment) or has no noticeable effect on the external output of the generated program image (e.g., it changes the format of the listing file, or causes debug information to be generated), is outside the scope of these coding guidelines.

Selecting translation-time options is part of the configuration management for a project.

Your author has never seen a set of coding guidelines that apply to make-files (apart from layout conventions) and would suggest that work on such a set is long overdue.

— the mechanism by which C programs are invoked for use by a data-processing system; 11

Commentary

The standard specifies the behaviors of C constructs. Specifying mechanisms for executing the generated program images serves no useful purpose at the level of abstraction at which the standard operates. Whichever mechanism is used, for a hosted implementation, the standard requires that a function called `main` be called by the execution environment.

program startup 150

Common Implementations

The output generated by a translator is usually written to a file. To indicate that this file can be executed, as a program, it might be given the extension *.exe* (under Microsoft Windows), or have its execute-bit set (under a POSIX-compliant operating systems such as Linux). Existing practices for invoking a program include giving the program name on the command line, clicking on icons, and having the program automatically executed on computer startup.

freestanding environment startup 155

In a freestanding environment the program image may be stored in read-only memory at a location that the host processor jumps to when it is initialized (which usually happens by default when power is first applied). The act of switching on, or resetting, is the mechanism for invoking the program.

---

input data mechanism transformed

— the mechanism by which input data are transformed for use by a C program; 12

Commentary

The standard is not concerned with how data is represented on media (which may be held on a hard disk, paper tape, or any other media) or an interactive device. It is the implementation's responsibility to map data from the bits held on a storage device to the input values they represent to a C program.

Coding Guidelines

Programs that want to access devices at a level below that specified by the C Standard are outside the scope of these coding guidelines.

Example

Specifying that a file is to be opened in text mode will cause the input to be treated as a series of lines. How lines are represented by the host file system is outside the scope of the C Standard.

end-of-line representation 224

— the mechanism by which output data are transformed after being produced by a C program; 13

Commentary

The Committee recognized that a program image may be executing within an encompassing environment. How this environment transforms data after it has been output by a program is outside the scope of the C Standard. The standard specifies an intended external effect for operations that perform output. It does not specify a *final resting place* for this external effect. It may be characters appearing on a display device, or bits being written to a storage device, or many other possibilities.

Common Implementations

At the host environment level (operating system) the sequences of bytes output by a C program are rarely modified until they reach the lower-levels of device drivers. Bytes sent over serial links may have parity bits added to them, blocks of bytes written to media may include file system information, and so on.

Coding Guidelines

Programs that are concerned with how output data are transformed once it has been generated by a program image are outside the scope of these coding guidelines.

14 1) This International Standard is designed to promote the portability of C programs among a variety of data-processing systems. footnote 1

Commentary

The Rationale puts the case very well:

Rationale

C code can be portable. Although the C language was originally born with the UNIX operating system on the DEC PDP-11, it has since been implemented on a wide variety of computers and operating systems. It has also seen considerable use in cross-compilation of code for embedded systems to be executed in a free-standing environment. The C89 Committee attempted to specify the language and the library to be as widely implementable as possible, while recognizing that a system must meet certain minimum criteria to be considered a viable host or target for the language.

C code can be non-portable. Although it strove to give programmers the opportunity to write truly portable programs, the C89 Committee did not want to force programmers into writing portably, to preclude the use of C as a "high-level assembler:" the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between strictly conforming program and conforming program (§4).

Avoid "quiet changes." Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible consistent with its other goals, the C89 Committee avoided changes that quietly alter one valid program to another with different semantics, that cause a working program to work differently without notice. In important places where this principle is violated, both the C89 Rationale and this Rationale point out a QUIET CHANGE.

A standard is a treaty between implementor and programmer. Some numerical limits were added to the Standard to give both implementors and programmers a better understanding of what must be provided by an implementation, of what can be expected and depended on to exist. These limits were, and still are, presented as minimum maxima (that is, lower limits placed on the values of upper limits specified by an implementation) with the understanding that any implementor is at liberty to provide higher limits than the Standard mandates. Any program that takes advantage of these more tolerant limits is not strictly conforming, however, since other implementations are at liberty to enforce the mandated limits.

Keep the spirit of C. The C89 Committee kept as a major goal to preserve the traditional spirit of C. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles on which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be how the target machine's hardware does it rather than by a general abstract rule. An example of this willingness to live with what the machine does can be seen in the rules that govern the widening of char objects for use in expressions: whether the values of char objects widen to signed or unsigned quantities typically depends on which byte operation is more efficient on the target machine.

One of the goals of the C89 Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the C89 Committee introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single-precision if both operands are float rather than double.

At the WG14 meeting in Tokyo, Japan, in July 1994, the original principles were re-endorsed and the following new ones were added:

Support international programming. During the initial standardization process, support for internationalization[1025] was something of an afterthought. Now that internationalization has become an important topic, it should have equal visibility. As a result, all revision proposals shall be reviewed with regard to their impact on internationalization as well as for other technical merit.

Codify existing practice to address evident deficiencies. Only those concepts that have some prior art should be accepted. (Prior art may come from implementations of languages other than C.) Unless some proposed new feature addresses an evident deficiency that is actually felt by more than a few C programmers, no new inventions should be entertained.

Minimize incompatibilities with C90 (ISO/IEC 9899:1990). It should be possible for existing C implementations to gradually migrate to future conformance, rather than requiring a replacement of the environment. It should also be possible for the vast majority of existing conforming programs to run unchanged.

Minimize incompatibilities with C++. The Committee recognizes the need for a clear and defensible plan for addressing the compatibility issue with C++. The Committee endorses the principle of maintaining the largest common subset clearly and from the outset. Such a principle should satisfy the requirement to maximize overlap of the languages while maintaining a distinction between them and allowing them to evolve separately.

The Committee is content to let C++ be the big and ambitious language. While some features of C++ may well be embraced, it is not the Committee's intention that C become C++.

Maintain conceptual simplicity. The Committee prefers an economy of concepts that do the job. Members should identify the issues and prescribe the minimal amount of machinery that will solve the problems. The Committee recognizes the importance of being able to describe and teach new concepts in a straight-forward and concise manner.

During the revision process, it was important to consider the following observations:

- Regarding the 11 principles, there is a trade-off between them-none is absolute. However, the more the Committee deviates from them, the more rationale will be needed to explain the deviation.
- There had been a very positive reception of the standard from both the user and vendor communities.
- The standard was not considered to be broken. Rather, the revision was needed to track emerging and/or changing technologies and internationalization requirements.
- Most users of C view it as a general-purpose high-level language. While higher-level constructs can be added, they should be done so only if they don't contradict the basic principles.
- There are a good number of useful suggestions to be found from the public comments and defect report processing.

Areas to which the Committee looked when revising the C Standard included:

- Incorporate AMD1.

- Incorporate all Technical Corrigenda and records of response.
- Current defect reports.
- Future directions in current standard.
- Features currently labeled obsolescent.
- Cross-language standards groups work.
- Requirements resulting from JTC 1/SC 2 (character sets).
- The evolution of C++.
- The evolution of other languages, particularly with regard to interlanguage communication issues.
- Other papers and proposals from member delegations, such as the numerical extensions Technical Report which was proposed by J11.
- Other comments from the public at large.
- Other prior art.

C++

No intended purpose is stated by the C++ Standard.

Coding Guidelines

What are coding guidelines designed to promote?

0 coding guidelines introduction

Rev 14.1 The first paragraph on page one of a coding guidelines document shall state the purpose of those guidelines and the benefits expected to accrue from adherence to them.

---

15 It is intended for use by implementors and programmers.

Commentary

One argument, used by the Committee, against the use of a formal definition language for specifying the requirements, in the standard, was that programmers would have difficulty understanding it. Given the small number of copies of the document actually sold by standards bodies, this seems to be a moot point. This situation may change with C99 thanks to a standards organization in the USA being willing to sell electronic copies at a reasonable price.

Although written using English prose, the wording of the standard is highly stylized. Readers need to become familiar with the conventions used if they are to correctly interpret its contents. The ISO directives also require that:

> *To achieve this objective, the International Standard shall*
>
> . . .
>
> — *be comprehensible to qualified persons who have not participated in its preparation.*

ISO Directives, part 3

During the initial development of the C Standard by the ANSI committee, the idea of the document being a *treaty* between implementor and developer was voiced by many members of that committee. The Rationale discusses this issue. Although many members of the ISO C committee also hold this view of a standard being a treaty, there are some members who view the document as being a specification.

14 treaty

Coding Guidelines

Who are the intended audience of these coding guidelines?

0 coding guidelines introduction

They are intended to be read by managers wanting to select a set of guideline recommendations applicable to their business model, authors of local coding guideline documents and training materials, and vendors

producing tools to enforce them. While some developers may chose to read these coding guidelines for educational purposes, there is no obvious cost/benefit justification for requiring all developers to systematically read them.

— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor; 16

Commentary

The standard does not require an implementation to fail to translate and execute a program that exceeds a size or complexity (how complexity might be measured is not specified) limit. Although such a requirement could increase program portability (they would have to be rewritten to reduce their size or complexity, making it more likely that they would be able to execute on a larger range of hosts). However, restricting those programs that may be translated for these reasons is counterproductive. Some translation limits effectively specify minimum bounds on program and data size that must be translated and executed by an implementation. However, they are lower, not upper limits.

limit 285
external identifiers

Common Implementations

It is not usually the size or complexity of programs that give translators problems. Optimizers like to keep all the information associated with a given function in memory while it is being translated. Functions that are large (complexity itself is rarely a problem) run the risk of having the translator run out of memory while generating machine code for them.

During execution all programs have a limit on the memory available to them to allocate for object storage. Most implementations will allocate storage until insufficient is available; problem execution usually fails in some way at this point.

Coding Guidelines

Programs that are too large or complex to be translated are fail-safe in the sense that developer attention is needed to solve the problem. Ensuring that programs do not run out of storage during execution, or that their execution terminates within a given time frame, are issues that are outside the scope of these coding guidelines. Some coding guideline documents address the storage capacity issue by prohibiting the use of constructs that prevent a program's total storage requirements from being known at translation time.[1026]

— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation. 17

Commentary

limit 285
external identifiers

This statement is not quite true. When deciding on values for the minimum translation limits, the C99 Committee had in mind a translation host with a total of 256 K of memory. The requirements needed to support a conforming implementation are considered to be a quality-of-implementation issue. Implementation vendors are left to respond to customer demands.

Common Implementations

Implementations have been created on hosts having a total memory size of 64 K.

2. Normative references

Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. 18

Commentary

A normative reference is the one that carries the same weight as wording in the standard itself. An informative reference is just that, informative.

Just as the Standard proper excludes all examples, footnotes, references, and informative annexes, this Rationale is not part of the Standard. The C language is defined by the Standard alone. If any part of this Rationale is not in accord with that definition, the Committee would very much like to be so informed.

The ISO Directives do not permit the *Normative references* clause to contain

- documents that are not publicly available,
- documents to which only informative reference is made, or
- documents that have merely served as references in the preparation of the standard.

These kinds of documents may be listed in a bibliography. Other normative documents, not listed here, can be created through the mechanism of Defect Reports (DRs) raised against wording in the existing standards document. A DR can be raised by a National Body (a country who is a P, Participating, member of SC22), the Project Editor, or the convener of WG14. 0 defect report

The C committee tries to deal with DRs during the meeting immediately following their submission. The response might be to agree that there is a problem with existing wording in the standard and to provide amended wording, or to say that the issue described is not considered a problem with the standard. The committee can choose to add additional material to the standard by issuing an Amendment (such an Amendment requires a new work item, which needs the support of five P member countries before it can go ahead). There was one Amendment for C90, dealing with wide character issues.

An ISO committee can also chose to issue Technical Reports (TRs). Work has started on a TR relating to C99 (the *Embedded C* Technical Report[657]). It deals with embedded systems issues (fixed-point types, differentiating different kinds of memory, saturation arithmetic, etc.). Up-to-date information on the C Standard can be found at the official Web site, http://www.open-std.org/jtc1/sc22/wg14/. Embedded C TR

Coding Guidelines

Referencing other documents, from within coding guidelines, could mean the reader has to spend significant time obtaining a copy of that reference. In many cases readers are unlikely to invest the effort needed to locate the referenced document.

Prior to C99 the cost of obtaining a copy of the C Standard was relatively high. The introduction of electronic distribution, and support from ANSI, has made a much lower-cost copy of the standard available. It is now realistic for guidelines to assume that their readers have access to a copy of the C Standard.

Rev 18.1 A coding guidelines document shall try to minimize references to other documents; if necessary, by including the relevant information in the guidelines document, perhaps in an annex.

Understanding the issues behind most DRs requires a close reading of the wording in the standard and usually involve situations that do not occur very often. In theory most DRs will not be of interest because use of constructs should not be relying on close readings of the standard. In practice usage of particular wording in the standard may be incidental, or the developer may not have read the wording closely at all.

The DRs raised against the C90 Standard are unlikely to have had any significant impact on programs (a few implementations had to change the way they handled constructs). It is hoped that the DRs raised against C99 follow this pattern. Authors of coding guidelines documents might like to periodically check the official log of DRs on the C Standard Web site, http://www.open-std.org/jtc1/sc22/wg14/.

A coding guidelines document is likely to be referred to by higher-level documents. The extent to which such documents follow the cost/benefit aims of these coding guidelines, or are even known to be effective, is unknown (although one study[1241] that experimentally looked at the consistency of ISO/IEC 15504 Software Process Improvement found some interesting results).

19 For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. Dated references

Commentary

Who knows what changes a future revision of a normative document might have. This sentence prevents revisions of some normative documents having an unintended impact on the interpretation of wording in the current version of the standard. This sentence is also explicitly stating a rule that is specified in the ISO directives.

C90

This sentence did not appear in the C90 Standard.

C++

1.2p1 *At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.*

The C++ Standard does not explicitly state whether later versions of standards do or do not apply. In the case of the C++ library, clause 17.3.1.4 refers to "the ISO C standard,", which could be read to imply that agreements based on the C++ Standard may reference either the C90 library or the C99 library. The C++ ISO Technical Report TR 19768 (C++ Library Extensions) includes support for the wide character library functionality that is new in C99, but does not include support for some of the type generic maths functions (some of these are the subject of work on a separate TR) or extended integer types. However, the current C++ Standard document effective references the C90 library.

However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. 20

Commentary

ISO rules require that every five years the Committee responsible for a standard investigate: should the standard be reconfirmed, should the standard be withdrawn, should the standard be revised. It typically seems to take around five years to produce and revise a language standard, creating a 10-year cycle; most other kinds of standards are either confirmed or are on a shorter cycle. Different standards can also be on different parts of their cycle. It usually takes at least 12 months for some kind of formal update to a standard to be adopted.

While parties may investigate the possibility of applying the most recent editions of standards, the timescales involved in formally adopting them are such that, unless there is a very important issue, the Committee may well decide to wait for the next revision of the standard to update the references.

C90

This sentence did not appear in the C90 Standard.

Coding Guidelines

It is unlikely that a revision of one of these standards will materially affect the C Standard. It is more likely that a revision of one of these standards will affect a developer's application domain. It is this wider issue which is outside the scope of these guidelines, that is likely to be of more importance to an application.

For undated references, the latest edition of the normative document referred to applies. 21

Commentary

The referenced standards will contain a date of publication. The C Standard requires that the most up-to-date version be applied.

C90

This sentence did not appear in the C90 Standard.

Coding Guidelines

Rev 21.1 A coding guidelines document shall specify which edition of the C Standard they refer to. They shall also state if any Addendums, Defect Reports, or Technical Reports are to be taken into account.

---

22 Members of ISO and IEC maintain registers of currently valid International Standards.

Commentary

The ISO Web site, http://www.iso.ch, is a good starting point for information.

Coding Guidelines

Putting coding guidelines documents under the same change control system as that used for source code is a good starting point for tracking revisions. However, the first priority should always be to make sure that the guideline recommendations are followed, not inventing new procedures to handle their change control.

---

23 ISO 31-11:1992, *Quantities and units— Part 11: Mathematical signs and symbols for use in the physical sciences and technology*. ISO 31-11

Commentary

This is Part 11 of a 13-part standard. Even though it is 27 pages long it is a nonexhaustive list of mathematical symbols. Part 1 of ISO 31, "Space and time", is a useful companion to ISO 8601. [27] ISO 8601

---

24 ISO/IEC 646, *Information technology— ISO 7-bit coded character set for information interchange*. ISO 646

Commentary

This standard assigns meanings to values in the range 0x0 to 0x7f. There are national variants of this standard (e.g., Ascii).

There are also 8-bit coded character set standards (i.e., ISO 8859–1 through ISO 8859–16) which assign meanings to the values 0x80 to 0xff (0x80–0x9f specify control codes and 0xa0 to 0xff additional graphics characters). ISO 8859–1 is commonly called *Latin-1* and covers all characters that occur in Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, and Swedish. This standard was recently replaced by ISO 8859–15, Latin-9, which added support for the Euro symbol and some forgotten French and Finnish letters. Values in the range 0x20 to 0x7e are always used to represent the invariant portion of ISO 646 (the so-called *International Reference Version*). ISO 8859

C++

ISO/IEC 10646–1:1993 Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane 1.2p1

---

25 ISO/IEC 2382–1:1993, *Information technology— Vocabulary — Part 1: Fundamental terms*. ISO 2382

Commentary

This is a reference to part 1 of a 27-part standard. The other 26 parts define the vocabulary for a wide range of computer-related areas.

C++

1.2p1 *ISO/IEC 2382 (all parts), Information technology — Vocabulary*

ISO 4217, *Codes for the representation of currencies and funds.* 26

Commentary

Quoting from its scope: "This International Standard provides the structure for a three letter alphabetic code and an equivalent three-digit numeric code for the representation of currencies and funds." Apart from an example in the library section, the contents of this standard are not used within the C Standard. However, a translator vendor may need to use this standard to implement a locale. The document is shorter than its 31 pages suggest. Half of it is written in French.

C++

There is no mention of this document in the C++ Standard.

ISO 8601 ISO 8601, *Data elements and interchange formats— Information interchange— Representation of dates and times.* 27

Commentary

This standard specifies the presentation format for dates and times. It also covers issues such as whether weeks start on Sunday or Monday, and which is week-1 of a year. The definition of terms is provided by ISO 31–1, "Space and time". A translator vendor may need to use this standard to implement a locale.

ISO 31-11 23

C++

There is no mention of this document in the C++ Standard.

ISO 10646 ISO/IEC 10646 (all parts), *Information technology— Universal Multiple-Octet Coded Character Set (UCS).* 28

Commentary

The ISO/IEC 10646 Standard uses a 32-bit representation, with the code positions divided into 128 groups of 256 planes with each plane containing 256 rows of 256 cells. An industrial consortium, known as Unicode http://www.unicode.org, developed a 16-bit encoding that corresponded exactly to plane zero (known as the Basic Multilingual Plane) of the 32-bit encoding used in ISO/IEC 10646. The two groups eventually merged their efforts and at the time of this writing the Unicode encoding uses the range 0x000000 to 0x10FFFF.

The supported characters do not just include letters, numbers, and symbols denoting words or parts of words, they also include symbols for non-words. For instance, *BLACK SPADE SUIT* (U+2660) ♠, *MUSIC NATURAL SIGN* (U+266E) ♮, *BLACK TELEPHONE* (U+260E) ☎, and *WHITE SMILING FACE* (U+263A) ☺.

__STDC_ISO_10646__ 2021 macro The conditionally defined macro __STDC_ISO_10646__ may contain information on the version of ISO/IEC 10646 supported by an implementation.

Common Implementations

Support for Unicode is more commonly seen than that for the full (or subset that is larger than Unicode) ISO/IEC 10646 specification. Both specify three encoding forms, UTF (in Unicode this is an acronym for *Unicode Transformation Format*, while in ISO/IEC 10646 it is *UCS Transformation Format*), of encoding characters:

universal character name syntax 815

1. As a 32-bit value directly holding the numeric value, *UTF-32* (this is the value used in UCNs).

2. As one, or two, 16-bit values (values in the range 0x0000–0xD7FF and 0xE000–0xFFFF goes in one, 0x010000–0x10FFFF goes in two; values in the range 0xD800–0xDFFF in the first 16 bits indicate that another value, in the range 0xDC00–0xDFFF, follows), *UTF-16*. UTF-16

3. As a sequence of one or more 8-bit values, *UTF-8*. The following list shows the encoding used for various ranges of characters. For multibyte sequences, the number of leading 1's in the first octet equals the number of octets in the sequence: UTF-8

```
U+00000000–U+0000007F: 0xxxxxxx
U+00000080–U+000007FF: 110xxxxx 10xxxxxx
U+00000800–U+0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U+00010000–U+001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U+00200000–U+03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U+04000000–U+7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

C++

ISO/IEC 10646–1:1993 Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane 1.2p1

ISO/IEC 10646:2003 is not divided into parts and the C++ Standard encourages the possibility of applying the most recent editions of standards. [19] Dated references

---

29 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). IEC 60559

Commentary

The term *IEEE floating point* is often heard. This usage came about because the original standards on this topic were published by the IEEE. This standard for binary floating-point arithmetic is what many host processors have been providing for over a decade. However, its use is not mandated by C99. See annex F.1 for a discussion of the relationship between this standard and other related floating-point standards from which it was derived.

C90

This standard did not specify a particular floating-point format, although the values given as an example for **<float.h>** were IEEE-754 specific (which is now an International Standard, IEC 60559).

C++

There is no mention of this document in the C++ Standard.

Common Implementations

Figueroa del Cid[422] provides an extensive discussion on what the phrase *support the IEEE floating-point standard* might be interpreted to mean.

The representation for binary floating-point specified in this standard is used by the Intel x86 processor family, Sun SPARC, HP PA-RISC, IBM PowerPC, HP–was DEC– Alpha, and the majority of modern processors (some DSP processors support a subset, or make small changes, for cost/performance reasons; while others have more substantial differences e.g., TMS320C3x[1341] uses two's complement). There is also a publicly available software implementation of this standard.[552]

Other representations are still supported by processors (IBM 390 and HP–was DEC– VAX) having an existing customer base that predates the publication the documents on which this standard is based. These representations will probably continue to be supported for some time because of the existing code that relies on it (the IBM 390 and HP–was DEC– Alpha support both their companies respective older representations and the IEC 60559 requirements).

Coding Guidelines

There is a common belief that once the IEC 60559 Standard has been specified all of its required functionality will be provided by conforming implementations. It is possible that a C program's dependencies on IEC 60559 constructs, which can vary between implementations, will not be documented because of this common, incorrect belief (the person writing documentation is not always the person who is familiar with this standard).

Like the C Standard the IEC 60559 Standard does not fully specify the behavior of every construct.[700] It also provides optional behavior for some constructs, such as when underflow is raised, and has optional constructs that an implementation may or may not make use of, such as double standard. C99 does not always provide a method for finding out an implementation's behavior in these optional areas. For instance, there are no standard macros describing the various options for handling underflow.

3. Terms, definitions, and symbols

C90

The title used in the C90 Standard was "Definitions and conventions".

For the purposes of this International Standard, the following definitions apply. 30

Commentary

These definitions override any that may appear in other standards documents (including ISO 2382). In some cases terms used in the standard have a meaning that is different from their *plain* English usage. For instance, in:

```
1  int x_1,
2      x_2;
3
4  void f(void)
5  {
6  int x_1;
7  }
```

scope 415 same the objects `x_1` and `x_2`, at file scope, are said to have the same scope. However, there is a region of the program text in which one of these identifiers, `x_1`, is not visible (because of a declaration of the same name in a nested block).

C++

1.3p1 *For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply.*

17p9 *The following subclauses describe the definitions (17.1), and method of description (17.3) for the library. Clause 17.4 and clauses 18 through 27 specify the contents of the library, and library requirements and constraints on both well-formed C++ programs and conforming implementations.*

Coding Guidelines

Some writers of coding guidelines find the definition of terms used in the C Standard hard to understand, or at least think that their readers might. This belief then becomes the rationale for creating a *less-experienced, reader friendly* definition of terms. Your author knows of no published, or unpublished, survey of the ease, or difficulty, developers have with various technical terms. Having two sets of definitions of terms is likely to lead to confusion. There is no evidence that one set of terms is better, or worse, than any other.

Rev 30.1 The definition of terms, as defined in the C Standard and standards referenced by it, shall be used in coding guideline documents.

---

31 Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. terms defined where

Commentary

In most cases the first use of a term is also where it is defined and hence where it usually appears in italic type.

C90

The fact that terms are defined when they appear "on the left side of a syntax rule" was not explicitly specified in the C90 Standard.

Coding Guidelines

A coding guidelines document cannot assume that its readers will start at the front and read each rule in turn. There are obvious advantages to collecting all terms, with a meaning specific to the guidelines, in an index or collecting them together in an annex. This is a usability issue that is outside the scope of these guidelines.

---

32 Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere.

Commentary

The C Standard is absolving itself of any similar terms that may be defined in any other standard.

Coding Guidelines

A coding guideline shall state that the terms defined by the C Standard are the ones that apply to itself.

---

33 Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382–1.

Commentary

Terms defined in the C Standard take precedence. If the term is not defined there, refer to ISO/IEC 2382–1 [25] ISO 2382 (Part II of ISO/IEC 2382 deals with mathematical and logical operations and is also a useful source of definitions). There has been discussion within the Committee on terms that are not defined by either document, but are technical in nature. In these cases the common dictionary usage has been claimed to be applicable. The ISO Directives specify that the two dictionaries *The Shorter Oxford English Dictionary* and *The Concise Oxford Dictionary*, provide the definitions of nontechnical words.

C++

> *For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply.* 1.3p1

The C++ Standard thus references all parts of the above standard. Not just the first part.

Coding Guidelines

While the above might appear to be a good sentence to include in a coding guidelines document, most developers are unlikely to have easy access to a copy of ISO/IEC 2382–1.

Rev 33.1 All technical terms used in a coding guidelines document shall be defined in that document.

Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31-11. 34

Commentary

The ISO/IEC 2382-II Standard deals with mathematical and logical operations but is not referenced by the C Standard. It is not known if there are any incompatibilities between this document and ISO 31-11.

3.1

access 35

access

⟨execution-time action⟩ to read or modify the value of an object

Commentary

While the behavior for most kinds of access are simple and easy to deduce, this innocent looking definition hides several dark corners. For instance, does an expression that multiplies an object by 1 modify that object? modify 37 includes cases Yes. Does accessing a bit-field that shares a storage unit with another bit-field also cause an access to that other bit-field? WG14 are looking into creating a more rigorous specification of what it means to access an object. A Technical Report may be published in due course (it will take at least two years). At the time of this writing WG14 has decided to wait until the various dark corners have been more fully investigated and the issues resolved and documented before making a decision on the form of publication.

reference 70 object The term *reference* is also applied to *objects*. The distinction between the two terms is that programs reference objects but access their values. An unevaluated expression may contain references to objects, but it never accesses them. The term *designate an object* is used in the standard. This term can involve a reference, an access, or both, or neither. In the following:

```
1  int *p = 0;
2  (*&*p);
```

the lvalue `(*&*p)` both accesses and references an object, namely the pointer object `p`, but it does not &* 1092 designate any object. The term *designate* also focuses attention on a particular object under discussion. For instance, in the expression `a[n]` there are the references `a`, `n`, and `a[n]`; the expression may or may not access any or all of `a`, `n`, and `a[n]`. But the discussion is likely to refer to the object designated by `a[n]`, not its component parts.

C++

In C++ the term *access* is used primarily in the sense of *accessibility*; that is, the semantic rules dealing with when identifiers declared in different classes and namespaces can be referred to. The C++ Standard has a complete clause (Clause 11, Member access control) dealing with this issue. While the C++ Standard also uses *access* in the C sense (e.g., in 1.8p1), this is not the primary usage.

Common Implementations

Many of these corner cases involve potential optimizations that translator vendors might like to perform. Some translators containing nontrivial optimizers provide options, selectable by developers, that control the degree to which optimizations are attempted. Optimizations that may change the behavior of a construct (from one choice of undefined, or unspecified behavior to another one) are usually only enabled at the higher, try harder levels of optimization.

Example

In the following:

```
1  int f(int param)
2  {
3  struct {
4          volatile unsigned int mem1:2;
5                   unsigned int mem2:4;
```

```
6           volatile unsigned int mem3:6;
7                    unsigned int mem4:1;
8          } mmap;
9
10  return mmap.mem2 += param;
11  }
```

the member mem2 may be placed in the same storage unit as mem1 and/or mem3. If the processor does not have bit-field extraction instructions, it will be necessary to generate code to load one or more bytes to obtain the value of mem2. If the member mem1 is contained in one of those bytes, does an access of mem2 also constitute an access to mem1?

Another issue is the number of accesses to a bit-field. Obtaining the value of mem3 requires a single, read, access. But how many accesses are needed to modify mem3? One to read the value, which is modified; a store back into mem3 may require another read (to obtain the value of the other members stored in the same storage unit; perhaps mem4 in the preceding example); the modified value of mem3 is inserted into the bit pattern read and the combined value written back into the storage unit. An alternative implementation may hang on to the value of mem4 so that the second read access is not needed.

36 NOTE 1 Where only one of these two actions is meant, “read” or “modify” is used.

37 NOTE 2 “Modify” includes the case where the new value being stored is the same as the previous value.

modify includes cases

Commentary

This specification only needs to be followed exactly when there is more than one access between sequence points, or for objects declared with the volatile storage class. As far as the developer is concerned, a modification occurs. As long as the implementation delivers the expected result, it is free to do whatever it likes.

Example

```
1   extern int glob;
2   extern volatile int reg;
3
4   void g(void)
5   {
6   reg *= 1; /*
7              * Value of reg looks as if it is unchanged, but because
8              * it is volatile qualified an access to the object is
9              * required.  This access may cause its value to change.
10             */
11  /*
12   * The following cannot be optimized to a single assignment.
13   */
14  reg=glob;
15  reg=glob;
16
17  glob += 0; /* Value of glob unchanged but it is still modified. */
18  /*
19   * glob modified twice between sequence points -> undefined behavior
20   */
21  glob = (glob += 0) + 4;
22  }
```

38 NOTE 3 Expressions that are not evaluated do not access objects.

Commentary

The term *not evaluated* here means *not required to be evaluated by the standard*. Implementations do not have complete freedom to decide what not to evaluate. Reasons why an expression may not be evaluated include:

- being part of a statement which is not executed,
- being part of a subexpression whose evaluation is conditional on other subexpressions within a full expression, and
- being an operand of the **`sizeof`** operator.

sizeof 1122 operand evaluated
sizeof 1123 operand not evaluated

If an implementation can deduce that the evaluation of an expression causes no side-effects, it can use the as-if rule to optimize away the generation of machine code (to evaluate the expression).

as-if rule 122

Common Implementations

The inverse of this rule is one area where translators that perform optimizations have to be careful. In:

```
1  a = b + c;
```

b and c are accessed to obtain their values for the + operator. An optimizer might, for instance, deduce that their sum had already been calculated and is still available in a register. However, using the value held in that register is only possible if it can be shown that not accessing b and c will not change the behavior of the program. If either were declared with the volatile qualifier, for instance, such an optimization could not be performed.

Coding Guidelines

Accessing an object only becomes important if there are side effects associated with that access; for instance, if the object is declared with the **`volatile`** qualifier. This issue is covered by guideline recommendations discussed elsewhere.

sequence 187.1 points all orderings give same value
code 91.1 shall affect output

Example

```
1  extern int i, j;
2  extern volatile int k;
3
4  void f(void)
5  {
6  /*
7   * Side effect of access to k occurs if the left operand
8   * of the && operator evaluates to nonzero.
9   */
10 if ((i == 2) && (j == k))
11    /* ... */ ;
12 /*
13  * In the following expression k appears to be read twice.
14  * Only one of the subexpressions will ever be evaluated.
15  * There is no unspecified or undefined behavior.
16  */
17 i = (j < 3 ? (k - j) : (j - k));
18 }
```

3.2

alignment

alignment 39

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

Commentary

In an ideal world there would be no alignment requirements. In practice the designers of some processors have placed restrictions on the fetching of variously sized objects from storage. The underlying reason for these restrictions is to simplify (reduce cost) and improve the performance of the processor. Some processors do not have alignment requirements, but may access storage more quickly if the object is aligned on a particular address boundary.

The requirement that a pointer to an object behave the same as a pointer to an array of that object's type forces the requirement that `sizeof(T)` be a multiple of the alignment of `T`. If two objects, having different types, have the same alignment requirements then their addresses will be located on the same multiple of a byte address. Objects of character type have the least restrictive alignment requirements, compared to objects of other types. Alignment and padding are also behind the assumptions that need to be made for the common initial sequence concept to work.

1165 additive operators pointer to object
558 pointer to void same representation and alignment as
1038 common initial sequence

The extent to which alignment causes implementation-defined or undefined behavior is called out for each applicable construct. Various issues relating to alignment were the subject of DR #074.

C++

Object types have alignment requirements (3.9.1, 3.9.2). The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type. 3.9p5

There is no requirement on a C implementation to document its alignment requirements.

Other Languages

Most languages hide such details from their users.

Common Implementations

Alignment is a very important issue for implementations. Internally it usually involves trade-offs between storage and runtime efficiency. Externally it is necessary to deal with parameter passing interfaces and potentially the layout of structure members. The C language is fortunate in that many system interfaces are specified in terms of C. This means that other languages have to interface to its way of doing things, not the other way around.

1413 alignment addressable storage unit

Optimal code generation for modern processors requires implementations to consider alignment issues associated with cache lines. Techniques for finding and using the optimal memory alignment for the objects used in a program is an active research area.[812]

The Motorola 56000[970] allows its modulo arithmetic mode to be applied to pointer arithmetic operations (used to efficiently implement a circular buffer; the wraparound effectively built into the access instruction). The one requirement is that the storage be aligned on an address boundary that is a power of two greater than or equal to the buffer size. So an array of 10 characters needs to be placed on a 16-byte boundary, while an array of 100 characters would need to be on a 128-byte boundary (assuming they are to be treated as circular buffers). Translators for such processors often provide an additional type specifier to enable developers to indicate this usage.

Motorola 56000

The Intel XScale architecture[629] has data cache lines that start on a 32-byte address boundary. Arranging for aggregate objects, particularly arrays, to have such an alignment will help minimize the number of stalls while the processor waits for data, and it also enables optimal use of the prefetch instruction. The Intel x86 architecture has no alignment requirement, but can load multibyte objects more quickly if appropriately aligned. RISC processors tend to have strict alignment requirements, often requiring that scalar objects be aligned on a byte boundary that is a multiple of their size.

Does the storage class or name space of an identifier affect its alignment?

Many implementations make no attempt to save storage, for objects having static storage duration, by packing them close together. It is often simpler to give them all the same, worst-case, alignment. For hosted

implementations potential savings (e.g., storage saved, time to access) may be small; there don't tend to be significant numbers of scalar objects with static storage duration. However, in those cases where freestanding environments have limited storage availability, vendors often go to great lengths to make savings. The alignment of objects having static storage duration is sometimes controlled by the linker, which may need to consider more systemwide requirements than a C translator (e.g., handling other languages or program image restrictions). For example, the C translator may specify that all objects start on an even address boundary; objects with automatic storage duration being placed on the next available even address after the previous object, but the linker using the next eighth-byte address boundary for objects.

linkers 140

A host's parameter-passing mechanism may have a minimum alignment requirement, for instance at least the alignment of **int**, and stricter alignment for types wider than an **int**. Minimizing the padding between different objects with automatic storage duration reduces the stack overhead for the function that defines them. It also helps to ensure they all fit within any short addressing modes available within the instruction set of the host processor. Unlike objects with static storage duration, most translators attempt to use an alignment that is appropriate to the type of the object.

The `malloc` function does not know the effective type assigned to the storage it allocates. It has to make worst-case assumptions and allocate storage based on the strictest alignment requirements.

The members of a structure type are another context where alignment requirements can differ from the alignment of objects having the same type in non-member contexts.

member 1421
alignment

Most processor instructions operate on objects having a scalar type and any alignment requirements usually only apply to scalar types. However, some processors contain instructions that operate on objects that are essentially derived types and these also have alignment requirements (e.g., the Pentium streaming SIMD instructions[626]). The multiples that occur in alignment requirements are almost always a power of two. On some processors the alignment multiple of a scalar type is the same as its size, in bytes.

derived type 525

Alignment requirements are not always constant during translation. Several implementations provide **#pragma** directives that control the alignment used by translators. A common vendor extension for controlling the alignment of structure members is the **#pack** preprocessor directive. `gcc` supports several extensions for getting and setting alignment information.

- The keyword **__alignof__**, which returns the alignment requirement of its argument, for instance, `__alignof__ (double)` might return 8.
- The keyword **__attribute__** can be used to specify an object's alignment at the point of definition:

```
1  int x __attribute__((aligned(16))) = 0;
```

Coding Guidelines

Alignment is an issue that can affect program resource requirements, program interoperability, and source code portability.

Alignment can affect the size of structure types. Two adjacent members declared with different scalar types may have padding inserted between them. Ordering members such that those with the same type are adjacent to each other can eliminate this padding (because the alignment of a scalar type is often a multiple of its size). If many instances of an object having a particular structure type are created a large amount of storage may be consumed and developers may consider it to be worthwhile investigating ways of making savings; for instance, by changing the default alignment settings, or by using a vendor-supplied **#pragma**.[1314] If alignment is changed for a subset of structures there is always the danger that a declaration in one translation unit will have an alignment that differs from the others. Adhering to the guideline recommendation on having a single point of declaration reduces the possibility of this occurring.

alignment 39

identifier 422.1
declared in one file

Some applications chose to store the contents of objects having structure types, in binary form, in files as a form of data storage. This creates a dependency on an implementation and the storage layout it uses for types. Other programs that access these data files, or even the same program after retranslation, need to

ensure that the same structure types have the same storage layout. This is not usually a problem when using the same translator targeting the same host. However, a different translator, or a different host, may do things differently. The design decisions behind the choice of data file formats is outside the scope of these coding guidelines.

1354 storage layout

Programs that cast pointers to point at different scalar types are making use of alignment requirements information, or lack of them. Porting a program which makes use of such casts to a processor with stricter alignment requirements, is likely to result in some form of signal being raised (most processors raise some form of exception when an attempt is made to access objects on misaligned addresses). The cost of modifying existing code to work on such a processor is one of the factors that should have been taken into account when considering the original decision to allow a guideline deviation permitting the use of such casts.

743 pointer to void converted to/from

Developers sometimes write programs that rely on different types sharing the same alignment requirements, e.g., **int** and **long**. There are cases where the standard guarantees identical alignments, but in general there are no such guarantees. Such usage is making use of representation information and is covered by a guideline recommendation.

560 alignment pointer to structures
569.1 representation information using

Example

Do all the following objects have the same alignment?

```
1  #include <stdlib.h>
2  /*
3   * #pragma pack
4   */
5
6  struct S_1 {
7              unsigned char mem_1;
8              int mem_2;
9              float mem_3;
10            };
11
12 static unsigned char file_1;
13 static int           file_2;
14 static float         file_3;
15
16 void f(unsigned char param_1,
17        int           param_2,
18        float         param_3)
19 {
20 unsigned char        local_1, *p_uc;
21 int                  local_2, *p_i;
22 float                local_3, *p_f;
23
24 p_uc = (unsigned char *)malloc(sizeof(unsigned char));
25 p_i = (int *)malloc(sizeof(int));
26 p_f = (float *)malloc(sizeof(float));
27 }
```

No, they need not, although implementations often chose to use the same alignment requirements for each scalar type, independent of the context in which it occurs. The `malloc` function cannot know the use to which the storage it returns will be put, so it has to make worst-case assumptions.

```
1  /*
2   * Declare a union containing all basic types and pointers to them.
3   */
4  union align_u {
5          char           c, *cp;
6          short               h, *hp;
7          int            i, *ip;
8          long           l, *lp;
```

```
        long long      ll, *llp;
        float          f, *fp;
        double         d, *dp;
        long double    ld, *ldp;
        void          *vp;
/*
 * Pointer-to function.  The standard does not define any generic
 * pointer types, like it does for pointer to void.  The wording
 * only requires the ability to convert.  There is no requirement
 * to be able to call the converted function pointer.
 */
        void         (*fv)(void);
        void         (*fo)();
        void         (*fe)(int, ...);
        };
/*
 * In the following structure type, the first member has type char.
 * The second member, a union type, will be aligned at least to the
 * strictest alignment of its contained types.  Assume that there
 * is no additional padding at the end of the structure declaration
 * than in the union.  Then we can calculate the strictest alignment
 * required by any object type (well at least those used to
 * define members of the union).
 */
struct align_s {
        char           c;
        union align_u  u;
        };

#define _ALIGN_  (sizeof(struct _align_s) - sizeof(union _align_u))
```

3.3

argument

argument 40

actual argument

actual parameter (deprecated)

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

Commentary

The terms *argument* and *parameter* are sometimes incorrectly interchanged. One denotes an expression at the point of call, the other the identifier defined as part of a function or macro definition.

C++

The C++ definition, 1.3.1p1, does not include the terms *actual argument* and *actual parameter*.

Coding Guidelines

The term *actual parameter* is rarely used. While the standard may specify its usage as being deprecated, it cannot control the terms used by developers. This term is not used within the standard.

Example

```
#define M(a) ((a) - (1))

int f(int b) /* The object b is the parameter. */
{
```

```
5   return b + 2;
6   }
7
8   void g(int x, int y)
9   {
10  x=f(1); /* The literal 1 is the argument. */
11  y=M(x); /* The value of x is the argument. */
12  }
```

3.4

41 **behavior** behavior

external appearance or action

Commentary

Common usage of the word *behavior* would enable it to be applied to all kinds of constructs. By providing this definition, the standard is narrowing down the range of possible uses to a specific meaning.

Rationale

The terms unspecified behavior, undefined behavior, and implementation-defined behavior are used to categorize the result of writing programs whose properties the Standard does not, or cannot, completely describe. The goal of adopting this categorization is to allow a certain variety among implementations which permits quality of implementation to be an active force in the marketplace as well as to allow certain popular extensions, without removing the cachet of conformance to the standard.

External appearances can take many forms. Interactive devices may display pixels, memory-mapped devices may open and close relays, or a processor may not be as responsive (because it is executing a program whose purpose is to consume processor resources). The C Standard sometimes fully specifies the intended behavior and the ordering of actions (although this is not always unique). But in only one case does it discuss the issue of how quickly a behavior occurs.

203 interactive device intent

Coding Guidelines

Developers tend to use the word *behavior* in its general, dictionary sense and include internal changes to the program state. While this usage is not as defined by the C Standard, there is no obvious advantage in trying to change this existing practice.

3.4.1

42 **implementation-defined behavior** implementation-defined behavior

unspecified behavior where each implementation documents how the choice is made

Commentary

In choosing between making the behavior of a construct implementation-defined or unspecified, the Committee had to look at the ability of translator vendors to be able to meaningfully document their implementations behavior. For instance, for assignment the order in which the operands are evaluated may depend on a range of different conditions decided on during code optimization. The only way of documenting this behavior being to supply documentation showing the data structures and algorithms used, an impractical proposition.

49 unspecified behavior
100 implementation document
1294 assignment operand evaluation order

C90

Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

The C99 wording has explicitly made the association between the terms implementation-defined and unspecified that was only implicit within the wording of the C90 Standard. It is possible to interpret the C90 definition as placing few restrictions on what an implementation-defined behavior might be. For instance, raising a signal or terminating program execution appear to be permitted. The C99 definition limits the possible behaviors to one of the possible behaviors permitted by the standard.

signed integer conversion 685
implementation-defined

C++

The C++ Standard uses the same wording, 1.3.5, as C90.

Other Languages

The C Standard is up front, and in general explicitly specifies the behavior that may vary between implementations. Other languages are not always so explicit. A major design aim for Java was to make it free of any implementation-defined behaviors. The intent is that Java programs exhibit the same behavior on all hosts. It will take time to see the extent to which this design aim can be achieved in practice.

Common Implementations

Many vendors list all of the implementation-defined behavior in an appendix of the user documentation.

Coding Guidelines

Some implementation-defined behaviors have no effect on the behavior of the abstract machine. For instance, the handling of the **`register`** storage-class specifier can affect execution time performance, but the program semantics remain unchanged. Some implementation-defined behaviors affect the execution time characteristics of a conforming program without affecting its output; for instance, the layout of structure members having a bit-field type.

bit-field 1409
addressable storage unit

The C Standard's definition of a strictly conforming program is based on the output produced by that program, not what occurs internally in the executing program image. Many coding guideline documents contain a blanket recommendation against the use of any implementation-defined behavior. This is a simplistic approach to guideline recommendations that is overly restrictive.

strictly conforming program 91
output shall not program image 141

Rev 42.1 Use of implementation-defined behavior shall be decided on a case-by-case basis. The extent to which the parameters of these cost/benefit decisions occur on a usage-by-usage based, a project-by-project basis, or a companywide-basis, is a management decision.

Coding guideline documents commonly recommend that a program's dependency on any implementation-defined behavior be documented. What purpose does such a guideline serve? Any well-written documentation would include information on implementation dependencies. The coding guidelines in this book do not aim to teach people how to write documentation. That said, the list of implementation-defined behaviors in annex J provides a good starting point.

Usage

Annex J.3 lists 97 implementation-defined behaviors.

EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right. 43

EXAMPLE high-order bit propagation

3.4.2

locale-specific behavior 44

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

locale-specific behavior

Commentary

The concept of locale was introduced by the C Standard. It has been picked up and extended by POSIX.[636]

Some of the execution time behavior that was implementation-defined in C90 has become locale-specific in C99. This allows programs that need to adapt to a locale to maintain a greater degree of standards conformance. Use of locale-specific behavior does not affect the conformance status of a program, while use of implementation-defined behavior means it cannot be strictly conforming (assuming that the program's output depends on it).

Rationale

While responding to a Defect Report filed against C89, the Committee came to realize that the term, "implementation-defined," was sometimes being used in the sense of "implementation must document" when dealing with locales. The term, "locale-specific behavior," already in C89, was then used extensively in C95 to distinguish those properties of locales which can appear in a strictly conforming program. Because the presence or absence of a specific locale is, with two exceptions, implementation-defined, some users of the Standard were confused as to whether locales could be used at all in strictly conforming programs.

A successful call to `setlocale` has side effects, known informally as "setting the contents of the current locale," which can alter the subsequent output of the program. A program whose output is altered only by such side effects— for example, because the decimal point character has changed— is still strictly conforming.

A program whose output is affected by the value returned by a call to `setlocale` might not be strictly conforming. If the only way in which the result affects the final output is by determining, directly or indirectly, whether to make another call to `setlocale`, then the program remains strictly conforming; but if the result affects the output in some other way, then it does not.

Common Implementations

A locale, other than `"C"`, for which there are many common implementations is Japanese.

The POSIX locale registry is slowly beginning to accumulate information on the locales of planet Earth. Both the Unix and Microsoft Windows host environments provide some form of interface to locale databases. It is these locales that C implementations usually provide a means of accessing.

Coding Guidelines

There are two locale issues that relate to these coding guidelines:

1. *The locale in which source code is translated.* Many coding guidelines relate to how developers extract information from source code. Having the source code written in the locale of the reader is likely to make this process easier. The issues involved in using identifiers that contain characters other than those in the basic execution character set are discussed elsewhere.
2. *The locale in which translated programs are executed.* Users want programs that adapt to their locales. The priority given to ensuring that software satisfies user's requirements is invariably much higher than that given to satisfying coding guidelines.

815 universal character name syntax
796 identifier UCN

Writing programs that depend on locale-specific behavior obviously reduces their portability to other locales. There is also the possibility that expected behaviors will not be available if the locale is changed. However, the issues involved in deciding when to use locale-specific behavior are outside the scope of these coding guideline subsections.

Implementations do not always fully document their handling of locale-specific behavior. Locales are a concept that is still evolving. To gain confidence in the behavior of an implementation, test programs need to be written to verify the behavior is as documented. Dealing with partial documentation is outside the scope of these coding guidelines.

45 EXAMPLE An example of locale-specific behavior is whether the **`islower`** function returns true for characters other than the 26 lowercase Latin letters.

EXAMPLE locale-specific behavior

Example

The character e-acute is a lowercase letter in a Latin-1 locale, but not the "C" locale.

3.4.3

undefined behavior

undefined behavior 46

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Commentary

conforming programs may depend on 103

EXAMPLE constraint violation and undefined behavior 148

The term *nonportable* is discussed elsewhere.

Although a sequence of source code may be an erroneous program construct, a translator is only required to issue a diagnostic message for a syntax violation or a constraint violation. The erroneous data can occur during translation, or during execution. For instance, division by zero within a constant expression or a division operator whose right operand had been assigned a zero value during the execution of a previous statement.

The C Standard does not contain any requirements for issuing diagnostics at execution time.

C90

Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements.

indeterminate value 75

trap representation 579

reading indeterminate value undefined behavior

unsigned char 571

pure binary

Use of an indeterminate value need not result in undefined behavior. If the value is read from an object that has **`unsigned char`** type, the behavior is unspecified. This is because objects of type **`unsigned char`** are required to represent values using a notation that does not support a trap representation.

Common Implementations

While the C Standard may specify that use of a construct causes undefined behavior, developers may have expectations of behavior or be unaware of what the C Standard has to say. Implementation vendors face customer pressure to successfully translate existing code. For this reason diagnostic messages are not usually generated at translation time when a construct causing undefined behavior is encountered.

The following are some of the results commonly seen when executing constructs exhibiting undefined behavior:

- *A signal is raised.* For instance, `SIGFPE` on divide by zero, or `SIGSEGV` when dereferencing a pointer that does not refer to an object.
- *The defined behavior of the processor occurs.* For instance, two's complement modulo rules for signed integer overflow.
- *The machine code generated as a result of a translation time decision is executed.* For instance, `i = i++;` may have been translated to the machine code `LOAD i; STORE i; INC i;` (instead of `INC i; LOAD i; STORE i;`, or some other combination of instructions).

Coding Guidelines

Developers are often surprised to learn that some construct, which they believed to have well-defined behavior, actually has undefined behavior. They often have clear ideas in their own heads of what the implementation behavior is in these cases and consider that to be the behavior mandated by the standard. This is an issue of developers' education and is outside the scope of these coding guidelines (although vendors of static analysis tools may consider it worthwhile issuing a diagnostic for uses of such constructs).

Usage

Annex J.2 lists 190 undefined behaviors.

---

47 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

undefined behavior possible

Commentary

This is only a list of possible behaviors; it is not intended to be a complete list. Behaving in a documented manner, plus issuing a diagnostic, is often considered to be the ideal case.

Common Implementations

The two most common ways of handling a construct, whose behavior is undefined, are to issue a diagnostic and to ignore it completely (the translator continuing to translate, or the execution environment delivering whatever result happens to occur). Some translators provide options that allow the developer to select the extent to which a translator will attempt to diagnose these constructs (e.g., the *-Wall* option of `gcc`). Very few implementations document any of their handling of undefined behaviors.

Some undefined behaviors often give consistent results, e.g., signed integer overflow, while in other cases the behavior is understood but the results are completely unpredictable. For instance, when accessing an object after its lifetime has ended, the common behavior is to access the storage previously assigned to that object, but the value held in that location is unpredictable.

Most diagnostics issued during program execution (most implementations issue a few during translation) are as a result of the program violating some host requirement in some way (for instance, a misaligned access to storage) and the host issuing a diagnostic prior to terminating program execution.

Coding Guidelines

There are a some undefined behaviors that give consistent results on many processors, for instance, the result of a signed integer overflow. Making use of such behavior is equivalent to making use of representation information, which is covered by a guideline recommendation.

569.1 representation information using

---

48 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

EXAMPLE undefined behavior

3.4.4

---

49 **unspecified behavior**

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

unspecified behavior

Commentary

The difference between unspecified (implementation-defined) and undefined is that the former applies to a correct program construct and correct data while the latter applies to an erroneous program construct or erroneous data. Are there any restrictions on possible translator behaviors? The following expresses the Committee's intent:

42 implementation-defined behavior

88 correct program

> This latitude does not extend as far as failing to translate the program.

Rationale

The wording was changed by the response to DR #247.

C90

Behavior, for a correct program construct and correct data, for which this International Standard imposes no requirements.

The C99 wording more clearly describes the intended meaning of the term *unspecified behavior*, given the contexts in which it is used.

C++

1.3.13 *behavior, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behavior occurs. [Note: usually, the range of possible behaviors is delineated by this International Standard.]*

This specification suggests that there exist possible unspecified behaviors that are not delineated by the standard, while the wording in the C Standard suggests that all possible unspecified behaviors are mentioned.

Common Implementations

Because there is no requirement to document the behavior in these cases, and the frequent difficulty of being able to say anything specific, most vendors say nothing.

Coding Guidelines

For implementation-defined behaviors, it is possible to read the documented behavior. For unspecified behaviors vendors do not usually document possible behaviors.

All nontrivial programs contain unspecified behavior; for instance, in the expression `b+c`, it is unspecified whether the object `b` is accessed before or after the object `c`. But, unless both objects are defined using the **`volatile`** qualifier, the order has no affect on a program's output.

type qualifier 1476 syntax

A blanket guideline recommendation prohibiting the use of any construct whose behavior is unspecified would be counterproductive. If the behavior of a construct is unspecified, but the behavior of the program containing it is identical for all of the different possibilities, the usage should be regarded as acceptable. How possible, different unspecified behaviors might be enumerated and the effects these behaviors have on the output of a program is left to developers to deduce. This issue is also covered by a guideline recommendation discussed elsewhere.

sequence 187.1 points all orderings give same value

Example

In an optimizing compiler the order of evaluation of expressions is likely to depend on context: What registers are free to store intermediate values; What registers contain previously calculated values that can be reused in the current expression? Documenting the behavior would entail describing all of the flow analysis, expression tree structure, and code optimization algorithms. In theory the order of evaluation, for each expression, could be deduced from this information. However, it would be impractical to carry out.

```
1  #include <stdio.h>
2
3  static int glob_1, glob_2;
4
5  int main(void)
6  {
7  if ((glob_1++, (glob_1+glob_2)) == (glob_2++, (glob_1+glob_2)))
8     printf("We may print this\n");
9  else
10    printf("or we may print this\n");
11 }
```

Usage

Annex J.1 lists 50 unspecified behaviors.

50 EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

3.5

51 **bit** bit

unit of data storage in the execution environment large enough to hold an object that may have one of two values

Commentary

Although it is the most commonly used representation in silicon-based processors, two-valued logic is not the most efficient form of representation. The most efficient radix, in terms of representation space (number of digits times number of possible values of each digit), is, e[555] (i.e., 2.718 . . .). The closest integral value to e is 3. While using a ternary representation maximizes the efficiency of representation, there are practical problems associated with its implementation.

The two states needed for binary representation can be implemented using a silicon transistor in its off (very low-voltage, high-current) and saturated (high-voltage, very low-current) states. Transistors in these two states consume very little power (voltage times current). Using transistors to implement a ternary representation would require the use of a third voltage (for instance, midway between low and high). At such a midpoint voltage, the current would also be mid-way between very low and high and the corresponding power consumption would be significantly higher than the off and saturated states. Power consumption, or rather the heat generated by it, is a significant limiting factor in processors built using transistors.

Vendors have chosen to trade-off efficiency of representation for the lower power consumption needed to prevent chips melting. Processors based on a binary representation are overwhelmingly used today. The definition of the C language reflects this fact.

C++

The C++ Standard does not explicitly define this term. The definition given in ISO 2382 is “Either of the digits 0 or 1 when used in the binary numeration system.”. 25 ISO 2382

Other Languages

The concept of a bit is a low-level one, intimately connected to the processor architecture. Many languages do not get explicitly involved in this level of detail. The language of Carbon-based processors (at least on planet Earth), DNA, uses a unit of storage that has one of four possible values (a *quyte*[41]): *A*, *C*, *G*, or *T*.

Common Implementations

Processors rarely provide a mechanism for referencing a particular bit in storage. The smallest unit of addressable storage is usually the byte. There are a few applications where the data is not byte-aligned and processors supporting some form of bit-level addressing are available.[1012] Automatic mapping of C source that performs its own manipulation of sequences of bits (using the bitwise operators) to use these processor instructions is not yet available. However, at least one research compiler[1432] provides support via what, to the developer, looks like a function-call interface. 53 byte addressable unit

52 NOTE It need not be possible to express the address of each individual bit of an object. bit address of

Commentary

The smallest object that is required to be addressable is one having character type. The number of bits in such an object is defined by the `CHAR_BIT` macro, which must have a value greater than or equal to 8. 515 character types 307 CHAR_BIT macro

Common Implementations

Processors rarely support bit addressing, although a few of them have instructions for extracting a sequence of bits (that is not a multiple of a byte) from a storage location.

3.6

byte
addressable unit

byte 53

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

Commentary

octet

A byte is traditionally thought of as occupying 8 bits. The technical term for a sequence of 8 bits is *octet*. POSIX[636] defines a byte as being an octet (i.e., 8 bits). These standards also define the terms *byte* and *character* independently of each other.

character 59 single-byte
CHAR_BIT 307 macro
char 477 hold any member of execution character set

There is something of a circularity in the C Standard's definition of byte and character. The macro `CHAR_BIT` defines the number of bits in a character type and is required to have a minimum value of 8. The definition of byte deals with data storage, while that for the **char** type deals with type.

This term first appeared in print in 1959.[176]

Other Languages

The DNA encoding used in Carbon based processors uses three *quytes*[41] (each of which has one of the four possible values *A*, *C*, *G*, or *T*) to form a *codon*. The 64 possible codons are used to represent different amino acids, which are used to make proteins, plus a representation for *stop*.

byte
address unique

NOTE 1 It is possible to express the address of each individual byte of an object uniquely. 54

Commentary

banked 559 storage

What is an address? In most cases it is taken to be the value representation held in a pointer object. In some cases a pointer object may not contain all the information needed to calculate an address. For instance, the IAR PICmicro compiler[612] provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size. On such a host the address of an object is the combination of the pointer value and information on the bank being referred to (which is encoded either in the instruction accessing the byte or a status register).

There is no requirement that the *addressable unit* be the address returned by the address-of operator, with the object as its operand. An implementation may choose to indirectly reference the actual storage used to hold an object's value representation via a lookup table, indexed by the value returned by the address-of, **&**, operator. This index value is then the *address* seen by programs.

object 570 contiguous sequence of bytes
object 765 point at each bytes of
value 573 copied using unsigned char
structure 1206 members later compare later
lifetime 451 of object
union type 531 overlapping members

Objects are made up of one or more bytes and it is possible to calculate values that point at any of them. The standard is therefore requiring that the individual bytes making up an object are uniquely identifiable. An implementation cannot hide the internal bits of an object. The ordering of bytes within an object and their relative addresses is not defined by the standard (although the relative order of structure members and array elements is defined). This requirement does not prevent more than one object from sharing the same address (i.e., their lifetimes may be different, or they may be members of a union type).

Common Implementations

word addressing

So-called *word addressed* processors only assign unique addresses to units of storage larger than 8 bits (for instance, 32-bit quantities, the word). Implementations for such processors have two choices: (1) define a byte to be the word size (e.g., the Motorola DSP56300 which has a 24 bit word[967] and the Analog Devices SHARC has a 32-bit word[30]), or (2) addresses of types smaller than a word use a different representation. Unless there is hardware support, the choice is usually to select the first option. With hardware support, some vendors chose option 2 (or to be exact they did not like option 1 and added support for a different representation, the offset of the byte within a word being encoded in additional representation bits). For instance, the Cray PVP[515] uses a 64-bit representation for pointers to character and **void** types and a 32-bit value representation for pointers to other types (a 64-bit object representation is used); a version of PR1ME Computers C compiler used 48-bit and 32-bit representations, respectively.

For the HP 3000 architecture, it was necessary to be extremely careful of conversions from byte addresses to word addresses and vice versa because of negative stack addressing, which makes byte addresses ambiguous.

The original Hewlett-Packard MPE processor (not the later models based on PA-RISC) was word addressed, using 16-bit words (128 K bytes). Byte addresses were signed from -64 K to +64 K, the sign depending on the current setting of the DL and Z registers. This scheme made it possible to access all the storage using byte addresses, but it created an ambiguity in the interpretation of a byte address value (this could only be resolved by knowing the settings of two processor registers).

Some processors are *word addressed* and don't use all the available representation bits. For instance, the Data General Nova[319] used only 15 of a possible 16 bits to address storage. The convention of using the additional bit to represent one of the two 8-bit quantities within a 16-bit word was adopted as a software solution to a hardware limitation (i.e., two software routines were provided to read and write individual bytes in storage, using addresses constructed using this convention).

The C Standard recognizes that not all pointer types have the same representation requirements.

562 alignment pointers

Coding Guidelines

Developers often assume more about the properties of the addresses of bytes than the C Standard requires of an implementation support; for instance, the extent to which bytes are allocated at similar addresses (an important consideration if performing relational operations on those addresses). The extent to which it is possible to make assumptions about the addresses of bytes is discussed elsewhere.

570 object contiguous sequence of bytes

Example

```
#include <stdio.h>

extern short glob;

void f(void)
{
int working_g = glob;
/*
 * Write out bytes, starting from least significant byte.
 */
for (int g_index=0; g_index < sizeof(glob); g_index++)
   {
   unsigned char uc = (working_g & UCHAR_MAX);
   fwrite(&uc, 1, 1, stdout);
   working_g >>= CHAR_BIT;
   }

/*
 * Will the following generate least or most significant
 * byte orderings?  Could be same or different from above.
 */
fwrite(&glob, sizeof(glob), 1, stdout);
}
```

---

55 NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined.

Commentary

Implementations where, for instance, the first four bits of a byte are separated from the second four bits by a single bit that is not part of that byte are not permitted. The type **`unsigned char`** is specified to use all the bits in its representation. The representational issues of this contiguous sequence of bits are discussed elsewhere.

573 value copied using unsigned char
595 value representation
574 object representation

The number of bits must be at least 8, the minimum value of the `CHAR_BIT` macro.

307 CHAR_BIT macro

Common Implementations

At the hardware level, storage can use one or more parity bits for error detection and correction (more than one bit is required for this). These additional bits are not visible through the conventional storage access mechanisms. Some hardware platforms do provide methods for accessing the bits in storage at the chip level.

---

low-order bit

The least significant bit is called the *low-order bit*; 56

Commentary

This defines the term *low-order bit*.

---

high-order bit

the most significant bit is called the *high-order bit*. 57

Commentary

This defines the term *high-order bit*.

3.7

---

character <abstract>

character 58

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

Commentary

This is the abstract definition of the term *character* (it is a very minor rewording of the definition given in ISO/IEC 2382–1:1993). The C-specific sense of the term *character* is given in the following (standard) sentence.

Characters are not created as stand-alone entities. Each is part of a larger whole, a character set. There are a large number of character sets (see Fischer[425] for the history until 1968), one for almost ever human language in the world. A character set provides another means of interpreting sequences of bits.

glyph

Characters are the smallest components of written languages that can have semantic value. The visible appearance of a character when it is displayed is called a *glyph* (there are often many different possible glyphs that can be used to represent the same character). A single character may be representable in a single byte (usually an alphabetic character) or may require several bytes (a multibyte character, often representing what English speakers would call a symbol). A repertoire (set) of glyphs is called a font (see Figure 792.17).

character 59
single-byte
multibyte 60
character
basic source 221
character set

The C Standard defines the basic character set that it requires an implementation to support.

C++

The C++ Standard does not define the term *character*; however, it does reference ISO/IEC 2382. Part 1, Clause 01.02.11, defines *character* using very similar wording to that given in the C Standard. The following might also be considered applicable.

17.1.2 *in clauses 21, 22, and 27, means any object which, when treated sequentially, can represent text. The term does not only mean* **`char`** *and* **`wchar_t`** *objects, but any value that can be represented by a type that provides the definitions provided in these clauses.*

3.7.1

---

character single-byte

character 59

single-byte character ⟨C⟩ bit representation that fits in a byte

Commentary

byte 53
addressable unit

This is the C-specific definition of the term *character*. It is a little circular in that byte is defined in terms of a character. The definition of the term *character*, referring to it in the abstract sense, is given in the previous sentence. Members of the basic source character set are required to fit in a byte. By definition any other character that fits in a byte is a single-byte character.

basic source 221
character set

C++

The C++ Standard does not define the term *character*. This term has different meanings in different application contexts and human cultures. In a language that supports overloading, it makes no sense to restrict the usage of this term to a particular instance.

Coding Guidelines

The terms character and byte are often used interchangeably by C developers. Changing developers' existing terminological habits may be very difficult, but this is not a reason why coding guideline documents should be sloppy in their use of terminology. Separating the two concepts can be helpful in highlighting potential problems associated with assuming particular representations for members of character sets (usually Ascii).

3.7.2

60 **multibyte character** multibyte character

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

Commentary

Multibyte characters are a means of representing characters from those character sets that contain more members than can be represented in a byte.[875] Somewhat confusingly the term *extended character set* is used by the C Standard to denote all supported characters, not just the extended characters. 216 extended character set 215 extended characters

A multibyte character is usually made up of a sequence of bytes that can be generated by pressing keys on the commonly available keyboards. There is not usually an obvious correspondence between the sequence of byte values and the numeric value of a member of the execution character set (such a correspondence does exist for a wide character), but there is an algorithm (often specified using a finite state machine) for converting them to this execution character set value. 243 multibyte character state-dependent encoding

Note: The use of the term *character* in the C Standard excludes multibyte characters, unless explicitly stated otherwise. This convention is also followed in the non-C Standard material in this book.

Common Implementations

The sequence of bytes in a multibyte character often has no relationship to what a developer types on a keyboard. For instance, at one end of the scale UTF-8 is unrelated to keystrokes. At the other end, many European keyboards have *dead keys* so that the single-byte characters *a-grave* or *i-circumflex* might be typed as *grave* followed by *a* or *circumflex* followed by *i*.

The most commonly used encoding methods include ISO 2022, EUC (Extended Unix Code), Big Five, Shift-JIS, and ISO 10646. Lunde[875,876] covers East Asian characters and their representations and encodings in great detail. 243 ISO 2022

61 NOTE The extended character set is a superset of the basic character set.

Commentary

The definition of the terms *basic character set* and *extended character set* implies that they are disjoint sets. 215 basic character set 216 extended character set

3.7.3

62 **wide character** wide character

bit representation that fits in an object of type **`wchar_t`**, capable of representing any character in the current locale

Commentary

A wide character can be thought of as the execution-time representation of a multibyte character. It is a pattern of bits held in an object, of type `wchar_t`, much like a character is a pattern of bits. The `wchar_t` type 60 multibyte character

often contains more than one byte, so it is capable of representing many more values. The bit representation of a particular character, held in an object of `wchar_t` type, can vary between locales.

Wide characters are best suited to be used when the numeric values of the characters are of importance, or when a fixed-size object is needed to manipulate character data.

C++

The C++ Standard uses the term *wide-character literal* and *wide-character sequences*, 17.3.2.1.3.3, but does not define the term *wide character*.

2.13.2p2 *A character literal that begins with the letter L, such as `L'x'`, is a wide-character literal.*

Common Implementations

On many implementations the `wchar_t` type usually occupies 16 bits. So, up until recently, it was capable of ISO 10646 28 being used to hold the values assigned to characters by the Unicode Consortium.

3.8

constraint

constraint 63

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

Commentary

These are the constructs that a conforming translator must detect and issue a diagnostic for, if encountered during translation of the source. Constraint and syntax violations are the only kinds of construct, defined by the standard, for which an implementation is required to issue a diagnostic.

In C, constraints only appear in subclauses that have the heading Constraints. These appear in Clause 6, shall 82 Language, but not Clause 7, Library. These language constraints are specified by use of the word *shall* within one of these subclauses. Violating a requirement specified using the word *shall* that appears within another shall 84 outside constraint kind of subclause causes undefined behavior.

C++

The C++ Standard does not contain any constraints; it contains diagnosable rules.

1.3.14 well-formed program *a C++ program constructed according to the syntax rules, diagnosable semantics rules, and the One Definition Rule (3.2).*

However, the library does use the term constraints.

17.4.3 Constraints on programs *This subclause describes restrictions on C++ programs that use the facilities of the C++ Standard Library.*

17.4.4p1 *This subclause describes the constraints upon, and latitude of, implementations of the C++ Standard library.*

But they are not constraints in the C sense of requiring a diagnostic to be issued if they are violated. Like C, the C++ Standard does not require any diagnostics to be issued during program execution.

Coding Guidelines

There is a common, incorrectly, held belief that diagnostics are issued for violations of any requirement contained within the C Standard. Diagnostics need only be issued for violations of requirements contained within a Constraints clause. Educating developers of the differences between constraints and the other forms of behavior described in the standard is an issue that is outside the scope of these coding guidelines.

Usage

There are 134 instances of the word *shall* in a Constraints clause (out of 588 in the complete standard). This places a lower bound on the number of constraints that can be violated (some uses of *shall* describe more than one kind of construct).

3.9

---

64 **correctly rounded result** correctly rounded result

representation in the result format that is nearest in value, subject to the ~~effective~~current rounding mode, to what the result would be given unlimited range and precision

Commentary

In Figure 64.1 the value b is midway between the representable values x and y. The representable value it will round to depends on the sign of its value, whether the rounding mode is round-to-even (least significant bit not set), round-to-zero, or round-to ±infinity. The values a and c have a *nearest* value to round to, if that rounding mode is in effect. [352] FLT_ROUNDS

`DBL_MAX*(1+DBL_EPSILON)` is *nearest* to `DBL_MAX` (in the Euclidean sense) but it is rounded to infinity.

The result referred to here is the result of a single arithmetic operation, not the result of an expression containing more than one operation. For instance, the expression `a+b-c` consists of the operation `a+b`, whose result then has `c` subtracted from it. If `a=1`, `b=1E-25`, and `c=1`, the correctly rounded result (assuming IEC 60559 single-precision) of `a+b` is `1`, and after the subtraction operation the final result of the complete expression is `0` (the expression can be rewritten to return the result `1E-25`).

The notation used to indicate rounded arithmetic operations is to place a circle around the operator; for instance:

$$x \oplus y = round(x + y) \tag{64.1}$$

where $x + y$ denotes the exact mathematical result. Based on the preceding example it is obvious that, in general:

$$a \oplus b \ominus c \neq a \ominus c \oplus b \tag{64.2}$$

The issue of correctly rounded results for some of the functions defined in the header **`<math.h>`** is discussed by Lefèvre and Muller.[827]

The wording was changed by the response to DR #286.

C++

This term is not defined in the C++ Standard (the term *rounded* only appears once, when discussing rounding toward zero).

Common Implementations

Implementations that don't always round correctly include some Cray processors and the IBM S/360 (truncates).

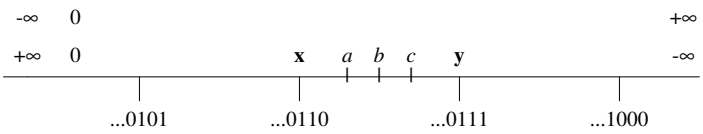


Figure 64.1: Some exactly representable values and three values (a, b, and c) that are not exactly representable.

Some implementations use a different rounding process when performing I/O than when performing floating-point operations. For instance, should the call `printf("%.0f", 2.5)` produce the same correctly rounded result (as output on `stdout`) as the value stored in an equivalent assignment expression?

Parks[1058] discusses methods for generating test cases that stress various boundary conditions that occur when rounding floating-point values.

Coding Guidelines

The fact that the term *correctly rounded result* can denote different values under different circumstances is something that should be included in the training of developers' who write code that uses floating-point types.

Obtaining the correctly rounded result of an addition or subtraction operation requires an additional bit in the significand (as provided by the IEC 60559 guard bit) to hold the intermediate result.[500] When multiplying two p bit numbers the complete $2p$ bits of the intermediate result need to be generated, before producing the final correctly rounded, p bit, result. A floating-point calculation that frequently occurs, for instance, in the evaluation of the determinant of a matrix is:

```
1  float d, w, x, y, z;
2
3  /* Use casts in case the minimum evaluation format is float. */
4
5  d = ((double)w) * x - ((double)y) * z;
```

If the relative precision of the type **`double`** is at least twice that of the type **`float`**, then the result will be correctly rounded (implementations where the types **`float`** and **`double`** have the same precision are not uncommon).

3.10

diagnostic message

diagnostic message

message belonging to an implementation-defined subset of the implementation's message output

Commentary

There is no requirement that the diagnostic message say anything of consequence. An implementation could choose to play a tune. Market forces usually dictate that the quality of most implementation diagnostic messages be more informative.

Common Implementations

Most implementations attempt to localize the construct being diagnosed. Localization usually occurs to the extent of giving a line number in the source code and an offset within that line. In some cases, for instance during macro expansion, the exact line number or character offset within a line may not be clear-cut.

macro replacement

All vendors like to think that their diagnostic messages are meaningful to developers. Diagnostic messages sometimes include a reference to what is considered to be the appropriate section of the C Standard.

Example

```
1  void f(int valu)
2  {
3  int log;
4
5  /* ... */
6  if (valu == 99)
7     {
8     int loc;
9     /* ... */
10    }
```

```
11
12  valu = loc;
13  /*
14   * What diagnostic is applicable to the previous statement:
15   *
16   *    o loc not declared,
17   *    o statement valu=loc should occur before prior closing brace,
18   *    o Previous closing brace should appear after this statement,
19   *    o Identifier log misspelled as loc.
20   */
21  }
```

3.11

66 **forward reference** forward reference

reference to a later subclause of this International Standard that contains additional information relevant to this subclause

Commentary

The references contained within a forward reference are not always exhaustive, neither are all applicable forward references always cited.

C++

C++ does not contain any forward reference clauses. However, the other text in the other clauses contain significantly more references than C99 does.

3.12

67 **implementation** implementation

particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

Commentary

The Committee tried to be as general as possible. They did not want to tie down how C programs might be processed to obtain the output required by the abstract machine.

It is not necessary to move to a different O/S or processor to use a different implementation. Each new release of a translator is a different implementation, even each invocation of a translator with different options enabled, or use of a different set of libraries, is a different implementation.

C++

The C++ Standard does not provide a definition of what an implementation might be.

Common Implementations

The software running in the translation environment is usually called a compiler. The software running in the execution environment is usually called the runtime system; it may also include an interpreter for the code generated by the compiler.

3.13

68 **implementation limit** implementation limit

restriction imposed upon programs by the implementation

Commentary

The Committee recognized that there are limits in the real world. The intent of calling them out is to highlight to the developer where they might occur and, by specifying values to give a base to work from. 273 environmental limits

Common Implementations

translation limits 276

All implementations have limits of various kinds. Even if very large amounts of memory and swap space are available, given a sufficiently large program, it will not be sufficient. Implementations also place limits on the range and precision of representable values. In this case there are efficiency issues; it would be possible to provide arbitrary ranges and precision. The greater the range and precision that must be handled, the longer it takes to perform an operation.

Coding Guidelines

Having a program keep within all implementation limits increases its portability. However, the extent to which portability to hosts is likely to impose limits, that a program exceeds, is an issue that needs to be decided by management. Specific implementation issues are discussed, in these coding guidelines, where they occur in the C Standard.

3.14

object

object

region of data storage in the execution environment, the contents of which can represent values

Commentary

object 570
contiguous sequence of bytes
value representation 595
object representation 574

Objects are created by definitions and calls to the library memory-allocation functions. The region of data storage for objects is a contiguous sequence of bytes. Representation of values within the region of data storage allocated to objects is discussed in great detail later in the standard.

C++

While an *object* is also defined, 1.8p1, to be a region of storage in C++, the term has many other connotations in an object-oriented language.

Other Languages

What C calls objects are often called variables in other languages.

Coding Guidelines

Many developers use the term *variable* to refer to what the C Standard calls an object. The introduction of the term *object-oriented* has meant that few developers use the term *object* in the C sense. Trying to change developer terminology in this case is liable to lead to confusion, and to be of little use (apart from being technically correct).

Example

```
#include <stdlib.h>

extern int glob;    /* Declaration of a named object. */

void f(long param)  /* A parameter is also an object. */
{
typedef short S_17; /* No storage created, not an object. */

float * p_f = (float *)malloc(sizeof(float)); /* Create an unnamed object. */
}
```

The response to DR #042 (question 2) specified that the bytes from which an object is composed need not correspond to any type declared in the program. For instance, a contiguous sequence of elements within an array can be regarded as an object in its own right. Nonoverlapping portions of an array can be regarded as objects in their own right.

```
#include <stdlib.h>
#include <string.h>
```

```
3
4   extern int N;
5
6   void DR_042(void)
7   {
8   void *p = malloc(2*N); /* Allocate an object. */
9
10     {
11     char (*q)[N] = p; /*
12                        * The object pointed to by p may be interpreted
13                        * as having type (char [2][N]) when referenced
14                        * through q.
15                        */
16     memcpy(q[1], q[0], N);
17     }
18     {
19     char *r = p; /*
20                   * The object pointed to by p may be interpreted as
21                   * having type (char [2*N]) when referenced through r.
22                   */
23     memcpy(r+N, r, N);
24     }
25  }
```

70 NOTE When referenced, an object may be interpreted as having a particular type; see 6.3.2.1.

reference object

Commentary

A reference that is also an access requires the bits making up an object to be interpreted as a value. This requires that it be interpreted as having a particular type. A reference that does not interpret the contents of an object, for instance an argument to the `memcpy` library function, does not need to interpret it as having a particular type. The issue of the type of an object, when it is referenced, is dealt with in detail elsewhere.

35 access
723 particular type
948 effective type

C++

The properties of an object are determined when the object is created. 1.8p1

This referenced/creation difference, compared to C, is possible in C++ because it contains the **`new`** and **`delete`** operators (as language keywords) for dynamic-storage allocation. The type of the object being created is known at the point of creation, which is not the case when the `malloc` library function is used (one of the reasons for the introduction of the concept of effective type in C99).

948 effective type

Coding Guidelines

A guideline recommendation dealing with the types that may be used to reference a given object is discussed elsewhere.

949.1 object effective type shall stay the same

Example

```
1   static int glob;
2
3   void f(void)
4   {
5   unsigned char *p_uc = (unsigned char *)&glob;
6
7   glob = 3;  /* Object glob interpreted to have type int. */
8
```

```
9   *p_uc = 3; /* Same object interpreted as an array of unsigned char. */
10  }
```

3.15

parameter

parameter 71

formal parameter

formal argument (deprecated)

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

Commentary

The term *formal argument* is rarely used. The term *formal parameter* is sometimes seen in documents written by people with a mathematically oriented computer science education.

Coding Guidelines

argument 40

The terms *argument* and *parameter* are sometimes incorrectly interchanged in discussions between developers. The context can often be used by the listener to deduce the intended meaning.

Example

```
1   #define FUNC_MACRO(x, y) /* Two parameters. */
2
3   void f(int param_1)      /* One parameter.  */
4   { /* ... */ }
5
6   int g(param_2)           /* One parameter.  */
7   int param_2;
8   { /* ... */ }
```

3.16

recommended practice

recommended practice 72

specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations

Commentary

The Recommended practice subsections have no more status than any other statement in the C Standard that is not mandatory.

C90

The *Recommended practice* subsections are new in C99.

C++

C++ gives some recommendations inside "[Note: . . .]", but does not explicitly define their status (from reading C++ Committee discussions it would appear to be non-normative).

Other Languages

Implementation Advice

Optional advice given to the implementer. The word "should" is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

Common Implementations

The C99 Standard is still too new to have any experience with how closely the new concept of recommended practice is followed.

Coding Guidelines

The extent to which *recommended practice*s will be followed by implementations is unknown. The specific instances are discussed when they occur.

3.17

---

73 **value** value

precise meaning of the contents of an object when interpreted as having a specific type

Commentary

For instance, the bits making up an object could be interpreted as an integer value, a pointer value, or a floating-point value. The definition of the type determines how the contents are to be interpreted. 1352 declaration interpretation of identifier

A literal also has a value. Its type is determined by both the lexical form of the token and its numeric value. 835 integer constant type first in list

C++

The value representation of an object is the set of bits that hold the value of type T. 3.9p4

Coding Guidelines

This definition separates the ideas of representation and value. A general principle behind many guidelines is that making use of representation information is not cost effective. The C Standard does not provide many guarantees that any representation is fixed (in places it specifies that two representations are the same). 569.1 representation information using

Example

```
1  #include <stdio.h>
2
3  union {
4        float mem_1;
5        int mem_2;
6        char *mem_3;
7        } x = {1.234567};
8
9  int main(void)
10 {
11 /*
12  * Interpret the same bit pattern using various types.
13  * The values output might be: 1.234567, 1067320907, 0x3f9e064b
14  */
15 printf("%f, %d, %p\n", x.mem_1, x.mem_2, x.mem_3);
16 }
```

3.17.1

implementation-defined value

implementation-defined value 74

unspecified value where each implementation documents how the choice is made

Commentary

unspecified value 76

Implementations are not required to document any unspecified value unless it has been specified as being implementation-defined. The semantic attribute denoted by an implementation-defined value might be applicable during translation (e.g., `FLT_EVAL_METHOD`), or only during program execution (e.g., the values assigned to `argv` on program startup).

FLT_EVAL_METHOD 354
argv 171
values

C90

Although C90 specifies that implementation-defined values occur in some situations, it never formally defines the term.

C++

The C++ Standard follows C90 in not explicitly defining this term.

Coding Guidelines

symbolic name 822

Implementation-defined values can vary between implementations. In some cases the C Standard defines a symbol (usually a macro name) to have certain properties. The key to using symbolic names is to make use of the property they denote, not the representation used (which includes the particular numerical value, as well as the bit pattern used to represent that value). For instance, a comparison against `UCHAR_MAX` should not be thought of as a comparison against the value `255` (or whatever its value happens to be), but as a comparison against the maximum value an object having **`unsigned char`** type can have. In some cases the result of an expression containing a symbolic name can still be considered to have a property. For instance, `UCHAR_MAX-3` might be said to represent the symbolic value having the property of being three less than the maximum value of the type **`unsigned char`**.

Example

```
#include <limits.h>

int int_max_div_10 = INT_MAX / 10;    /* 1/10th of the maximum representable int. */
int int_max_is_even = INT_MAX & 0x01; /* Testing for a property using representation information. */
```

3.17.2

indeterminate value

indeterminate value 75

either an unspecified value or a trap representation

Commentary

object 461
initial value indeterminate

This is the value objects have prior to being assigned one by an executing program. In practice it is a conceptual value because, in most implementations, an object's value representation makes use of all bit patterns available in its object representation (there are no spare bit patterns left to represent the indeterminate value).

unspecified value 76
trap representation 579
reading is undefined behavior

Accessing an object that has an unspecified value results in unspecified behavior. However, accessing an object having a trap representation can result in undefined behavior.

C++

Objects may have an indeterminate value. However, the standard does not explicitly say anything about the properties of this value.

4.1p1

. . . , or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.

Common Implementations

A few execution time debugging environments tag storage that has not had a value stored into it so that read accesses to it cause a diagnostic to be issued.

Coding Guidelines

Many coding guideline documents contain wording to the effect that "indeterminate value shall not be used by a program." Developers do not intend to use such values and such usage is a fault. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

0 guidelines not faults

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  int int_loc;     /* Initial value indeterminate. */
6  unsigned char uc_loc;
7
8  /*
9   * The reasons behind the different status of the following
10  * two assignments is discussed elsewhere.
11  */
12 glob = int_loc; /* Indeterminate value, a trap representation. */
13 glob = uc_loc;  /* Indeterminate value, an unspecified value.  */
14 }
```

3.17.3

76 **unspecified value** unspecified value

valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance

Commentary

Like unspecified behavior, unspecified values can be created by strictly conforming programs. Making use of such a value is by definition dependent on unspecified behavior.

49 unspecified behavior

Coding Guidelines

In itself the generation of an unspecified value is usually harmless. However, a coding guideline's issue occurs if program output changes when different unspecified values, chosen from the set of values possible in a given implementation, are generated. In practice it can be difficult to calculate the affect that possible unspecified values have on program output. Simplifications include showing that program output does not change when different unspecified values are generated, or a guideline recommendation that the construct generating an unspecified value not be used. A subexpression that generates an unspecified value having no affect on program output is dead code.

49 unspecified behavior

190 dead code

Example

```
1  extern int ex_f(void);
2
3  void f(void)
4  {
```

```
5   int loc;
6   /*
7    * If a call to the function ex_f returns a different value each
8    * time it is invoked, then the evaluation of the following can
9    * yield a number of different possible results.
10   */
11  loc = ex_f() - ex_f();
12  }
```

NOTE An unspecified value cannot be a trap representation. 77

Commentary

correct program 88 Unspecified values can occur for correct program constructs and correct data. A trap representation is likely to raise an exception and change the behavior of a correct program.

3.18

$\lceil x \rceil$ 78

ceiling of *x*: the least integer greater than or equal to *x*

Commentary

ISO 31-11 23 The definition of a mathematical term that is not defined in ISO 31-11.

EXAMPLE $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2. 79

3.19

floor $\lfloor x \rfloor$ 80

floor of *x*: the greatest integer less than or equal to *x*

Commentary

ISO 31-11 23 The definition of a mathematical term that is not defined in ISO 31-11.

EXAMPLE $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3. 81

conformance

4. Conformance

Commentary

In the C90 Standard this header was titled *Compliance*. Since this standard talks about conforming and strictly conforming programs it makes sense to change this title. Also, from existing practice, the term *Conformance* is used by voluntary standards, such as International Standards, while the term *Compliance* is used by involuntary standards, such as regulations and laws.

SC22 had a Working Group responsible for conformity and validation issues, WG12. This WG was formed in 1983 and disbanded in 1989. It produced two documents: *ISO/IEC TR 9547:1988— Test methods for programming language processors – guidelines for their development and procedures for their approval* and *ISO/IEC TR 10034:1990— Guidelines for the preparation of conformity clauses in programming language standards*.

shall In this International Standard, “shall” is to be interpreted as a requirement on an implementation or on a program; 82

Commentary

How do we know which is which? In many cases the context in which the *shall* occurs provides the necessary information. Most usages of *shall* apply to programs and these commentary clauses only point out those cases where it applies to implementations.

The extent to which implementations are required to follow the requirements specified using *shall* is affected by the kind of subclause the word appears in. Violating a *shall* requirement that appears inside a subsection headed *Constraint* clause is a constraint violation. A conforming implementation is required to issue a diagnostic when it encounters a violation of these constraints.

84 shall outside constraint
63 constraint

The term *should* is not defined by the standard. This word only appears in footnotes, examples, recommended practices, and in a few places in the library. The term *must* is not defined by the standard and only occurs once in it as a word.

1622 EXAMPLE compatible function prototypes

C++

The C++ Standard does not provide an explicit definition for the term *shall*. However, since the C++ Standard was developed under ISO rules from the beginning, the default ISO rules should apply.

84 ISO shall rules

Coding Guidelines

Coding guidelines are best phrased using "shall" and by not using the words "should", "must", or "may".

Usage

The word *shall* occurs 537 times (excluding occurrences of *shall not*) in the C Standard.

---

83 conversely, "shall not" is to be interpreted as a prohibition.

Commentary

In some cases this prohibition requires a diagnostic to be issued and in others it results in undefined behavior. An occurrence of a construct that is the subject of a *shall not* requirement that appears inside a subsection headed *Constraint* clause is a constraint violation. A conforming implementation is required to issue a diagnostic when it encounters a violation of these constraints.

84 shall outside constraint
63 constraint

Coding Guidelines

Coding guidelines are best phrased using *shall not* and by not using the words *should not*, *must not*, or *may not*.

Usage

The phrase *shall not* occurs 51 times (this includes two occurrences in footnotes) in the C Standard.

---

84 If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined.

shall outside constraint

Commentary

This C sentence brings us onto the use of ISO terminology and the history of the C Standard. ISO use of terminology requires that the word *shall* implies a constraint, irrespective of the subclause it appears in. So under ISO rules, all sentences that use the word *shall* represent constraints. But the C Standard was first published as an ANSI standard, ANSI X3.159-1989. It was adopted by ISO, as ISO/IEC 9899:1990, the following year with minor changes (e.g., the term Standard was replaced by International Standard and there was a slight renumbering of the major clauses; there is a `sed` script that can convert the ANSI text to the ISO text), but the *shall*s remained unchanged.

ISO shall rules

If you, dear reader, are familiar with the ISO rules on *shall*, you need to forget them when reading the C Standard. This standard defines its own concept of constraints and meaning of *shall*.

C++

This specification for the usage of *shall* does not appear in the C++ Standard. The ISO rules specify that the meaning of these terms does not depend on the kind of normative context in which they appear. One implication of this C specification is that the definition of the preprocessor is different in C++. It was essentially copied verbatim from C90, which operated under different *shall* rules :-O.

84 ISO shall rules

Coding Guidelines

Many developers are not aware that the C Standard's meaning of the term *shall* is context-dependent. If developers have access to a copy of the C Standard, it is important that this difference be brought to their attention; otherwise, there is the danger that they will gain false confidence in thinking that a translator will issue a diagnostic for all violations of the stated requirements. In a broader sense educating developers about the usage of this term is part of their general education on conformance issues.

Usage

The word *shall* appears 454 times outside of a Constraint clause; however, annex J.2 only lists 190 undefined behaviors. The other uses of the word *shall* apply to requirements on implementations, not programs.

---

undefined behavior indicated by

Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. 85

Commentary

Failure to find an explicit definition of behavior could, of course, be because the reader did not look hard enough. Or it could be because there was nothing to find, implicitly undefined behavior. On the whole the Committee does not seem to have made any obvious omissions of definitions of behavior. Those DRs that have been submitted to WG14, which have later turned out to be implicitly undefined behavior, have involved rather convoluted constructions. This specification for the omissions of an explicit definition is more of a catch-all rather than an intent to minimize wording in the standard (although your author has heard some Committee members express the view that it was never the intent to specify every detail).

shall 84 outside constraint

The term *shall* can also mean undefined behavior.

C++

The C++ Standard does not define the status of any omission of explicit definition of behavior.

Coding Guidelines

Is it worth highlighting omissions of explicit definitions of behavior in coding guidelines (the DRs in the record of response log kept by WG14 provides a confirmed source of such information)? Pointing out that the C Standard does not always fully define a construct may undermine developers' confidence in it, resulting in them claiming that a behavior was undefined because they could find no mention of it in the standard when a more thorough search would have located the necessary information.

Example

The following quote is from Defect Report #017, Question 19 (raised against C90).

DR #017

X3J11 previously said, "The behavior in this case could have been specified, but the Committee has decided more than once not to do so. [They] do not wish to promote this sort of macro replacement usage." I interpret this as saying, in other words, "If we don't define the behavior nobody will use it." Does anybody think this position is unusual?

Response

If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token, it is unspecified whether this macro name may be subsequently replaced. If the behavior of the program depends upon this unspecified behavior, then the behavior is undefined.

For example, given the definitions:

```
#define f(a) a*g

#define g(a) f(a)
```

the invocation:

```
f(2)(9)
```

results in undefined behavior. Among the possible behaviors are the generation of the preprocessing tokens:

```
2*f(9)
```

and

```
2*9*g
```

```
Correction
Add to subclause G.2, page 202:
-- A fully expanded macro replacement list contains a
function-like macro name as its last preprocessing token (6.8.3).
```

Subclause G.2 was the C90 annex listing undefined behavior. Different wording, same meaning, appears in annex J.2 of C99.

---

86 There is no difference in emphasis among these three;

Commentary

It is not possible to write a construct whose behavior is more undefined than another construct, simply because of the wording used, or not used, in the standard.

Coding Guidelines

There is nothing to be gained by having coding guideline documents distinguish between the different ways undefined behavior is indicated in the C Standard.

---

87 they all describe "behavior that is undefined".

---

88 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

correct program

Commentary

As pointed out elsewhere, any nontrivial program will contain unspecified behavior.

49 unspecified behavior

A wide variety of terms are used by developers to refer to programs that are not correct. The C Standard does not define any term for this kind of program.

Terms, such as *fault* and *defect*, are defined by various standards:

defect. See fault.

error. (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretical correct value or condition.

(2) Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification. This is not the preferred usage.

fault. (1) An accidental condition that causes a functional unit to fail to perform its required function.

(2) A manifestation of an error(2) in software. A fault, if encountered, may cause a failure. Synonymous with bug.

ANSI/IEEE Std 729–1983, IEEE Standard Glossary of Software Engineering Terminology

Error (1) A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action that results in software containing a fault. Examples

ANSI/AIAA R–013-1992, Recommended Practice for Software Reliability

include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.

Failure (1) The inability of a system or system component to perform a required function with specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of a functional unit to perform its required function. (3) A departure of program operation from program requirements.

Failure Rate (1) The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs.

Fault (1) A defect in the code that can be the cause of one or more failures. (2) An accidental condition that causes a functional unit to fail to perform its required function. Synonymous with bug.

Quality The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.

Software Quality (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, to conform to specifications. (2) The degree to which software possesses a desired combination of attributes. (3) The degree to which a customer or user perceives that software meets his or her composite expectations. (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Software Reliability (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time.

C90

This statement did not appear in the C90 Standard. It was added in C99 to make it clear that a strictly conforming program can contain constructs whose behavior is unspecified, provided the output is not affected by the behavior chosen by an implementation.

C++

1.4p2 *Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:*

— If a program contains no violations of the rules of this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute that program.

footnote 3 *"Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.*

Programs which have the status, according to the C Standard, of being *strictly conforming* or *conforming* have no equivalent status in C++.

Common Implementations

A program's source code may look correct when mentally executed by a developer. The standard assumes that C programs are correctly translated. Translators are programs like any other, they contain faults. Until the 1990s, the idea of proving the correctness of a translator for a commercially used language was not taken seriously. The complexity of a translator and the volume of source it contained meant that the resources

required would be uneconomical. Proofs that were created applied to toy languages, or languages that were so heavily subseted as to be unusable in commercial applications.

Having translators generate correct machine code continues to be very important. Processors continue to become more powerful and support gigabytes of main storage. Researchers continue to increase the size of the language subsets for which translators have been proved correct.[836, 1003, 1496] They have also looked at proving some of the components of an existing translator, `gcc`, correct.[1002]

Coding Guidelines

The phrase *the program is correct* is used by developers in a number of different contexts, for instance, to designate intended program behavior, or a program that does not contain faults. When describing adherence to the requirements of the C Standard, the appropriate term to use is *conformance*.

Adhering to coding guidelines does not guarantee that a program is correct. The phase *correct program* does not really belong in a coding guidelines document. These coding guidelines are silent on the issue of what constitutes correct data.

---

89 The implementation shall not successfully translate a preprocessing translation unit containing a **`#error`** preprocessing directive unless it is part of a group skipped by conditional inclusion.

#error terminate translation

Commentary

The intent is to provide a mechanism to unconditionally cause translation to fail. Prior to this explicit requirement, it was not guaranteed that a **`#error`** directive would cause translation to fail, if encountered, 1993 #error
although in most cases it did.

C90

C90 required that a diagnostic be issued when a **`#error`** preprocessing directive was encountered, but the translator was allowed to continue (in the sense that there was no explicit specification saying otherwise) translation of the rest of the source code and signal *successful translation* on completion.

C++

. . . , and renders the program ill-formed. 16.5

It is possible that a C++ translator will continue to translate a program after it has encountered a **`#error`** directive (the situation is as ambiguous as it was in C90).

Common Implementations

Most, but not all, C90 implementations do not successfully translate a preprocessing translation unit containing this directive (unless skipping an arm of a conditional inclusion). Some K&R implementations failed to translate any source file containing this directive, no matter where it occurred. One solution to this problem is to write the source as `??=error`, because a K&R compiler would not recognize the trigraph.

Some implementations include support for a `#warning` preprocessor directive, which causes a diagnostic 1993 #warning
to be issued without causing translation to fail.

Example

```
#if CHAR_BIT != 8
#error Networking code requires byte == octet
#endif
```

---

90 A *strictly conforming program* shall use only those features of the language and library specified in this International Standard.[2)]

strictly conforming program
use features of language/library

Commentary

In other words, a strictly conforming program cannot use extensions, either to the language or the library. A strictly conforming program is intended to be maximally portable and can be translated and executed by any conforming implementation. Nothing is said about using libraries specified by other standards. As far as the translator is concerned, these are translation units processed in translation phase 8. There is no way of telling apart user-written translation units and those written by third parties to conform to another API standard.

translation phase 8 139

Rationale

The Standard does not forbid extensions provided that they do not invalidate strictly conforming programs, and the translator must allow extensions to be disabled as discussed in Rationale §4. Otherwise, extensions to a conforming implementation lie in such realms as defining semantics for syntax to which no semantics is ascribed by the Standard, or giving meaning to undefined behavior.

C++

1.3.14 well-formed program

a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

The C++ term *well-formed* is not as strong as the C term *strictly conforming*. This is partly as a result of the former language being defined in terms of requirements on an implementation, not in terms of requirements on a program, as in C's case. There is also, perhaps, the thinking behind the C++ term of being able to check statically for a program being well-formed. The concept does not include any execution-time behavior (which strictly conforming does include). The C++ Standard does not define a term stronger than *well-formed*.

standard 1 specifies form and interpretation

The C requirement to use only those library functions specified in the standard is not so clear-cut for freestanding C++ implementations.

1.4p7

For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.4.1.3).

Other Languages

Most language specifications do not have as sophisticated a conformance model as C.

Common Implementations

All implementations known to your author will successfully translate some programs that are not strictly conforming.

Coding Guidelines

extensions 95.1 cost/benefit

This part of the definition of strict conformance mirrors the guideline recommendation on using extensions. Translating a program using several different translators, targeting different host operating systems and processors, is often a good approximation to *all implementations* (this is a tip, not a guideline recommendation).

---

strictly conforming program output shall not

It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit. 91

Commentary

The key phrase here is *output*. Constructs that do not affect the output of a program do not affect its conformance status (although a program whose source contains violations of constraint or syntax will never get to the stage of being able to produce any output). A translator is not required to deduce whether a construct affects the output while performing a translation. Violations of syntax and constraints must be diagnosed independent of whether the construct is ever executed, at execution time, or affects program output. These are extremely tough requirements to meet. Even the source code of some C validation suites did not meet these requirements in some cases.[682]

implementation validation 92

Coding Guidelines

Many coding guideline documents take a strong line on insisting that programs not contain any occurrence of unspecified, undefined, or implementation-defined behaviors. As previously discussed, this is completely unrealistic for unspecified behavior. For some constructs exhibiting implementation-defined behavior, a strong case can be made for allowing their use. The issues involved in the use of constructs whose behavior is implementation-defined is discussed in the relevant sentences.

49 unspecified behavior
42 implementation-defined behavior

The issue of programs exceeding minimum implementation limits is rarely considered as being important. This is partly based on developers' lack of experience of having programs fail to translate because they exceed the kinds of limits specified in the C Standard. Program termination at execution time because of a lack of some resource is often considered to be an application domain, or program implementation issue. These coding guidelines are not intended to cover this kind of situation, although some higher-level, application-specific guidelines might.

The issue of code that does not affect program output is discussed elsewhere.

190 redundant code

Cg 91.1 All of a programs translated source code shall be assumed to affect its output, when determining its conformance status.

---

92 The two forms of *conforming implementation* are hosted and freestanding.

implementation two forms

Commentary

Not all hardware containing a processor can support a C translator. For instance, a coffee machine. In these cases programs are translated on one host and executed on a completely different one. Desktop and minicomputer-based developers are not usually aware of this distinction. Their programs are usually designed to execute on hosts similar to those that translate them (same processor family and same kind of operating system).

A freestanding environment is often referred to as the *target* environment; the thinking being that source code is translated in one environment with the aim of executing it on another, the target. This terminology is only used for a hosted environment, where the program executes in a different environment from the one in which it was translated.

The concept of implementation-conformance to the standard is widely discussed by developers. In practice implementations are not perfect (i.e., they contain bugs) and so can never be said to be conforming. The testing of products for conformance to International Standards is a job carried out by various national testing

implementation validation

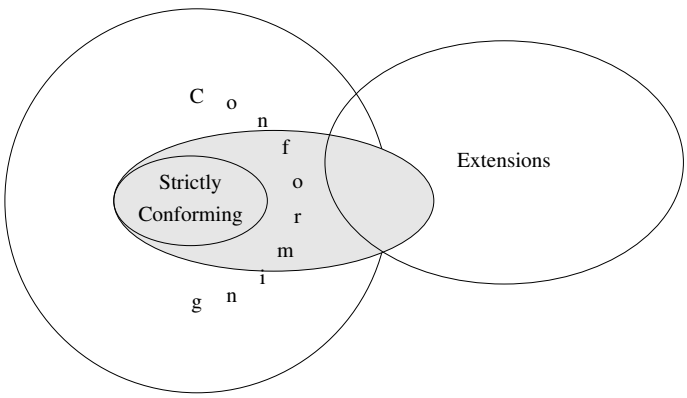


Figure 92.1: A conforming implementation (gray area) correctly handles all strictly conforming programs, may successfully translate and execute some of the possible conforming programs, and may include some of the possible extensions.

laboratories. Several of these testing laboratories used to be involved in testing software, including the C90 language standard (language implementation validation did not prove commercially viable and there are no longer any national testing laboratories offering this service). A suite of test programs was used to measure an implementation's handling of various constructs. An implementation that successfully processed the tests was not certified to be a conforming implementation but rather (in BSI's case): "This is to certify that the language processor identified below has been found to contain no errors when tested with the identified validation suite, and is therefore deemed to conform to the language standard."

Ideally, a validation suite should have the following properties:

- Check all the requirements of the standard.
- Tests should give the same results across all implementations (they should be strictly conforming programs).
- Should not contain coding bugs.
- Should contain a test harness that enables the entire suite to be compiled/linked/executed and a pass/fail result obtained.
- Should contain a document that explains the process by which the above requirements were checked for correctness.

There are two validation suites that are widely used commercially: Perennial CVSA (version 8.1) consists of approximately 61,000 test cases in 1,430,000 lines of source code, and Plum Hall validation suite (CV-Suite 2003a) for C contains 84,546 test cases in 157,000 lines of source. A study by Jones[682] investigated the completeness and correctness of the ACVS. Ciechanowicz[236] did the same for the Pascal validation suite.

Most formal validation concentrates on language syntax and semantics. Some vendors also offer automated expression generators for checking the correctness of the generated machine code (by generating various combinations of operators and operands whose evaluation delivers a known result, which is checked by translating and executing the generated program). Wichmann[1457] describes experiences using one such generator.

Other Languages

Most other standardized languages are targeted at a hosted environment.

Some language specifications support different levels of conformance to the standard. For instance, Cobol has three implementation levels, as does SQL (Entry, Intermediate, and Full). In the case of Cobol and Fortran, this approach was needed because of the technical problems associated with implementing the full language on the hosts of the day (which often had less memory and processing power than modern hand calculators).

The Ada language committee took the validation of translators seriously enough to produce a standard: ISO/IEC 18009:1999 *Information technology— Programming languages – Ada: Conformity assessment of a language processor*. This standard defines terms, and specifies the procedures and processes that should be followed. An Ada Conformity Assessment Test suite is assumed to exist, but nothing is said about the attributes of such a suite.

The POSIX Committee, SC22/WG15, also defined a standard for measuring conformance to its specifications. In this case they[620] attempted to provide a detailed top-level specification of the tests that needed to be performed. Work on this conformance standard was hampered by the small number of people, with sufficient expertise, willing to spend time writing it. Experience also showed that vendors producing POSIX test suites tended to write to the requirements in the conformance standard, not the POSIX standard. Lack of resources needed to update the conformance standard has meant that POSIX testing has become fossilized.

Java was originally designed to run in what is essentially a freestanding environment.

Common Implementations

The extensive common ground that exists between different hosted implementations does not generally exist within freestanding implementations. In many cases programs intended to be executed in a hosted environment are also translated in that environment. Programs intended for a freestanding environment are rarely translated in that environment.

---

93 A *conforming hosted implementation* shall accept any strictly conforming program.

conforming hosted implementation

Commentary

This is a requirement on the implementation. Another requirement on the implementation deals with limits. This requirement does not prohibit an implementation from accepting programs that are not strictly conforming.

276 translation limits
95 implementation extensions
90 strictly conforming program use features of language/library

A strictly conforming program can use any feature of the language or library. This requirement is stating that a conforming hosted implementation shall implement the entire language and library, as defined by the standard (modulo those constructs that are conditional).

C++

No such requirement is explicitly specified in the C++ Standard.

Example

Is a conforming hosted implementation required to translate the following translation unit?

```
1  int array1[5];
2  int array2[5];
3  int *p1 = &array1[0];
4  int *p2 = &array2[0];
5
6  int DR_109()
7  {
8  return (p1 > p2);
9  }
```

1209 relational pointer comparison undefined if not same object

It would appear that the pointers `p1` and `p2` do not point into the same object, and that their appearance as operands of a relational operator results in undefined behavior. However, a translator would need to be certain that the function `f_1` is called, that `p1` and `p2` do not point into the same object, and that the output of any program that calls it is dependent on it. Even in the case:

```
1  int f_2(void)
2  {
3  return 1/0;
4  }
```

a translator cannot fail to translate the translation unit unless it is certain that the function `f_2` is called.

---

94 A *conforming freestanding implementation* shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<float.h>**, **<iso646.h>**, **<limits.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, and **<stdint.h>**.

conforming freestanding implementation

Commentary

This is a requirement on the implementation. There is nothing to prevent a conforming implementation supporting additional standard headers, that are not listed here.

Complex types were added to help the Fortran supercomputing community migrate to C. They are very unlikely to be needed in a freestanding environment.

The standard headers that are required to be supported define macros, typedefs, and objects only. The runtime library support needed for them is therefore minimal. The header **<stdarg.h>** is the only one that may need runtime support.

C90

The header **<iso646.h>** was added in Amendment 1 to C90. Support for the complex types, the headers **<stdbool.h>** and **<stdint.h>**, are new in C99.

C++

1.4p7 *A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that include certain language-support libraries (17.4.1.3).*

17.4.1.3p2 *A freestanding implementation has an implementation-defined set of headers. This set shall include at least the following headers, as shown in Table 13:*

. . .

Table 13 C++ Headers for Freestanding Implementations

Subclause	Header(s)
18.1 Types	**<cstddef>**
18.2 Implementation properties	**<limits>**
18.3 Start and termination	**<cstdlib>**
18.4 Dynamic memory management	**<new>**
18.5 Type identification	**<typeinfo>**
18.6 Exception handling	**<exception>**
18.7 Other runtime support	**<cstdarg>**

The supplied version of the header ***<cstdlib>*** *shall declare at least the functions* `abort()`*,* `atexit()`*, and* `exit()` *(18.3).*

The C++ Standard does not include support for the headers **<stdbool.h>** or **<stdint.h>**, which are new in C99.

Common Implementations

String handling is a common requirement at all application levels. Some freestanding implementations include support for many of the functions in the header **<string.h>**.

Coding Guidelines

Issues of which headers must be provided by an implementation are outside the scope of coding guidelines. This is an application build configuration management issue.

---

implementation extensions

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[3)] 95

Commentary

The C committee did not want to ban extensions. Common extensions were a source of material for both C90 and C99 documents. But the Committee does insist that any extensions do not alter the behavior of other constructs it defines. Extensions that do not change the behavior of any strictly conforming program are sometimes called *pure extensions*.

An implementation may provide additional library functions. It is a moot point whether they are actual extensions, since it is not suggested that libraries supplied by third parties have this status. The case for calling them extensions is particularly weak if the functionality they provide could have been implemented by the developer, using the same implementation but without those functions. However, there is an established practice of calling anything provided by the implementation that is not part of the standard an extension.

Common Implementations

One of the most common extensions is support for inline assembler code. This is sometimes implemented by making the assembler code look like a function call, the name of the function being `asm`, e.g., `asm("ld r1, r2");`.

In the Microsoft/Intel world, the identifiers **`NEAR`**, **`FAR`**, and **`HUGE`** are commonly used as pointer type modifiers.

Implementations targeted at embedded systems (i.e., freestanding environments) sometimes use the `^` operator to select a bit from an object of a specified type. This is an example of a nonpure extension.

Coding Guidelines

These days vendors do not try to tie customers into their products by doing things different from what the C Standard specifies. Rather, they include additional functionality; providing extensions to the language that many developers find useful. Source code containing many uses of a particular vendor's extensions is likely to be more costly to port to a different vendor's implementation than source code that does not contain these constructs.

Many developers accumulated most of their experience using a single implementation; this leads them into the trap of thinking that what their implementation does is what is supported by the standard. They may not be aware of using an extension. Using an extension through ignorance is poor practice.

Use of extensions is not in itself poor practice; it depends on why the extension is being used. An extension providing functionality that is not available through any other convenient means can be very attractive. Use of a construct, an extension or otherwise, after considering all other possibilities is good engineering practice.

A commonly experienced problem with vendor extensions is that they are not fully specified in the associated documentation. Every construct in the C Standard has been looked at by many vendors and its consequences can be claimed to have been very well thought through. The same can rarely be said to apply to a vendor's extensions. In many cases the only way to find out how an extension behaves, in a given situation, is to write test cases.

Some extensions interact with constructs already defined in the C Standard. For instance, some implementations[20] define a type, using the identifier **`bit`** to indicate a 1-bit representation, or using the punctuator `^` as a binary operator that extracts the value of a bit from its left operand (whose position is indicated by the right operand).[717] This can be a source of confusion for readers of the source code who have usually not been trained to expect this usage.

Experience shows that a common problem with the use of extensions is that it is not possible to quantify the amount of usage in source code. If use is made of extensions, providing some form of documentation for the usage can be a useful aid in estimating the cost of future ports to new platforms.

Rev 95.1 The cost/benefit of any extensions that are used shall be evaluated and documented.

Dev 95.1 Use is made of extensions and:

- their use has been isolated within a small number of functions, or translation units,
- all functions containing an occurrence of an extension contain a comment at the head of the function definition listing the extensions used,
- test cases have to be written to verify that the extension operates as specified in the vendor's documentation. Test cases shall also be written to verify that use of the extension outside of the context in which it is defined is flagged by the implementation.

Some of the functions in the C library have the same name as functions defined by POSIX. POSIX, being an API-based standard (essentially a complete operating system) vendors have shown more interest in implementing the POSIX functionality.

Example

The following is an example of an extension, provided the VENDOR_X implementation is being used and the call to `f` is followed by a call to a trigonometric function, that affects the behavior of a strictly conforming program.

```
#include <math.h>

#if defined(VENDOR_X)
#include "vmath.h"
#endif

void f(void)
{
/*
 * The following function call causes all subsequent calls
 * to functions defined in <math.h> to treat their argument
 *  values as denoting degrees, not radians.
 */
#if defined(VENDOR_X)
switch_trig_to_degrees();
#endif
}
```

The following examples are pure extensions. Where might the coding guideline comments be placed?

```
/*
 * This function contains assembler.
 */
void f(void)
/*
 * This function contains assembler.
 */
{
/*
 * This function contains assembler.
 */
asm("make the, coffee"); /* How do we know this is an extension? */
} /* At least we can agree this is the end of the function. */

void no_special_comment(void)
{
asm("open the, biscuits");
}


void what_syntax_error(void)
{
asm wash up, afterwards
}

void not_isolated(void)
{
/*
 * Enough standard C code to mean the following is not isolated.
 */
asm wait for, lunch
}
```

footnote 2

2) A strictly conforming program can use conditional features (such as those in annex F) provided the use is guarded by a **`#ifdef`** directive with the appropriate macro.

Commentary

The definition of a macro, or lack of one, can be used to indicate the availability of certain functionality. The **#ifdef** directive providing a natural, language, based mechanism for checking whether an implementation supports a particular optional construct. The POSIX standard[636] calls macros, used to check for the availability (i.e., an implementations' support) of an optional construct, *feature test* macros.

feature test macro

C90

The C90 Standard did not contain any conditional constructs.

C++

The C++ Standard also contains optional constructs. However, testing for the availability of any optional constructs involves checking the values of certain class members. For instance, an implementation's support for the IEC 60559 Standard is indicated by the value of the member `is_iec559` (18.2.1.2).

29 IEC 60559

Other Languages

There is a philosophy of language standardization that says there should only be one language defined by a standard (i.e., no optional constructs). The Pascal and C90 Standard committees took this approach. Other language committees explicitly specify a multilevel standard; for instance, Cobol and SQL both define three levels of conformance.

C (and C++) are the only commonly used languages that contain a preprocessor, so this type of optional construct-handling functionality is not available in most other languages.

Common Implementations

If an implementation does not support an optional construct appearing in source code, a translator often fails to translate it. This failure invariably occurs because identifiers are not defined. In the case of optional functions, which a translator running in a C90 mode to support implicit function declarations may not diagnose, there will be a link-time failure.

Coding Guidelines

Use of a feature test macro highlights the fact that support for a construct is optional. The extent to which this information is likely to be already known to the reader of the source will depend on the extent to which a program makes use of the optional constructs. For instance, repeated tests of the `__STDC_IEC_559__` macro in the source code of a program that extensively manipulates IEC 60559 format floating-point values complicates the visible source and conveys little information. However, testing this macro in a small number of places in the source of a program that has a few dependencies on the IEC 60559 format is likely to provide useful information to readers.

2015 __STDC_IEC_ macro

Use of a feature test macro does not guarantee that a program correctly performs the intended operations; it simply provides a visual reminder of the status of a construct. Whether an **#else** arm should always be provided (either to handle the case when the construct is not available, or to cause a diagnostic to be generated during translation) is a program design issue.

Example

```
#include <fenv.h>

void f(void)
{
#ifdef __STDC_IEC_559__
   fesetround(FE_UPWARD);
#endif /* The case of macro not being defined is ignored. */

#ifdef __STDC_IEC_559__
   fesetround(FE_UPWARD);
#else
   #error Support for IEC 60559 is required
```

```
13  #endif
14
15  #ifdef __STDC_IEC_559__
16     fesetround(FE_UPWARD);
17  #else
18     /*
19      * An else arm that does nothing.
20      * Does this count as handling the alternative?
21      */
22  #endif
23  }
```

example __STDC_IEC_559__

For example: 97

```
#ifdef __STDC_IEC_559__ /* FE_UPWARD defined */
   /* ... */
   fesetround(FE_UPWARD);
   /* ... */
#endif
```

footnote 3

3) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard. 98

Commentary

If an implementation did reserve such an identifier, then its declaration could clash with one appearing in a strictly conforming program (probably leading to a diagnostic message being generated). The issue of reserved identifiers is discussed in more detail in the library section.

C++

The clauses 17.4.3.1, 17.4.4, and their associated subclauses list identifier spellings that are reserved, but do not specify that a conforming C++ implementation must not reserve identifiers having other spellings.

Common Implementations

In practice most implementation's system headers do define (and therefore could be said to reserve) identifiers whose spelling is not explicitly reserved for implementation use (see Table 1897.1). Many implementations that define additional keywords are careful to use the double underscore, `__`, prefix on their spelling. Such an identifier spelling is not always seen as being as readable as one without the double underscore. A commonly adopted renaming technique is to use a predefined macro name that maps to the double underscore name. The developer can always **`#undef`** this macro if its name clashes with identifiers declared in the source.

It is very common for an implementation to predefine several macros. These macros are either defined within the program image of the translator, or come into existence whenever one of the standard-defined headers is included. The names of the macros usually denote properties of the implementation, such as `SYSTYPE_BSD`, `WIN32`, `unix`, `hp9000s800`, and so on.

Identifiers defined by an implementation are visible via headers, which need to be included, and via libraries linked in during the final phase of translation. Most linkers have an *only extract the symbols needed* mode of working, which enables the same identifier name to be externally visible in the developers' translation unit and an implementation's library. The developers' translation unit is linked first, resolving any references to its symbol before the implementation's library is linked.

Coding Guidelines

Coding guidelines cannot mandate what vendors (translator, third-party library, or systems integrator) put in the system headers they distribute. Coding guideline documents need to accept the fact that almost no commercial implementations meet this requirement.

Requiring that all identifiers declared in a program first be **#undef**'ed, on the basis that they may also be declared in a system header, would be overkill (and would only remove previously defined macro names). Most developers use a suck-it-and-see approach, changing the names of any identifiers that do clash.

Identifier name clashes between included header contents and developer written file scope declarations are likely to result in a diagnostic being issued during translation. Name usage clashes between header contents and block scope identifier definitions may sometimes result in a diagnostic; for instance, the macro replacement of an identifier in a block scope definition resulting in a syntax or constraint violation.

Measurements of code show (see Table 98.1) that most existing code often contains many declarations of identifiers whose spellings are reserved for use by implementations. Vendors are aware of this usage and often link against the translated output of developer written code before finally linking against implementation libraries (on the basis that resolving name clashes in favour of developer defined identifiers is more likely to produce the intended behavior).

Whether the cost of removing so many identifier spellings potentially having informative semantics, to readers of the source, associated with them is less than the benefit of avoiding possible name clash problems with implementation provided libraries is not known. No guideline recommendation is given here.

Table 98.1: Number of developer declared identifiers (the contents of any header was only counted once) whose spelling (the notation *[a-z]* denotes a regular expression, i.e., a character between *a* and *z*) is reserved for use by the implementation or future revisions of the C Standard. Based on the translated form of this book's benchmark programs.

Reserved spelling	Occurrences
Identifier, starting with __, declared to have any form	3,071
Identifier, starting with *_[A-Z]*, declared to have any form	10,255
Identifier, starting with *wcs[a-z]*, declared to have any form	1
Identifier, with external linkage, defined in C99	12
File scope identifier or tag	6,832
File scope identifier	2
Macro name reserved when appropriate header is **#include**d	6
Possible macro covered identifier	144
Macro name starting with *E[A-Z]*	339
Macro name starting with *SIG[A-Z]*	2
Identifier, starting with *is[a-z]*, with external linkage (possibly macro covered)	47
Identifier, starting with *mem[a-z]*, with external linkage (possibly macro covered)	108
Identifier, starting with *str[a-z]*, with external linkage (possibly macro covered)	904
Identifier, starting with *to[a-z]*, with external linkage (possibly macro covered)	338
Identifier, starting with *is[a-z]*, with external linkage	33
Identifier, starting with *mem[a-z]*, with external linkage	7
Identifier, starting with *str[a-z]*, with external linkage	28
Identifier, starting with *to[a-z]*, with external linkage	62

99 A *conforming program* is one that is acceptable to a conforming implementation.[4)]

conforming program

Commentary

Does the conforming implementation that accepts a particular program have to exist? Probably not. When discussing conformance issues, it is a useful simplification to deal with possible implementations, not having to worry if they actually exist. Locating an actual implementation that exhibits the desired behavior adds nothing to a discussion on conformance, but the existence of actual implementations can be a useful indicator for quality-of-implementation issues and the likelihood of certain constructions being used in real programs (the majority of real programs being translated by an extant implementation at some point).

C++

The C++ conformance model is based on the conformance of the implementation, not a program (1.4p2). However, it does define the term *well-formed program*:

1.3.14 well-formed program

a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

Coding Guidelines

strictly conforming program
use features of language/library
Just because a program is translated without any diagnostics being issued does not mean that another translator, or even the same translator with a different set of options enabled, will behave the same way. A conforming program is acceptable to a conforming implementation. A strictly conforming program is acceptable to all conforming implementations.

The cost of migrating a program from one implementation to all implementations may not be worth the benefits. In practice there is a lot of similarity between implementations targeting similar environments (e.g., the desktop, DSP, embedded controllers, supercomputers, etc.). Aiming to write software that will run within one of these specific environments is a much smaller task and can produce benefits at an acceptable cost.

---

implementation document
An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions. 100

Commentary

The formal validation process carried out by BSI (in the UK) and NIST (in the USA), when they were in the language-implementation validation business, checked that the implementation-defined behavior was documented. However, neither organization checked the accuracy of the documented behavior.

C90

locale-specific behavior
Support for locale-specific characteristics is new in C99. The equivalent C90 constructs were defined to be implementation-defined, and hence were also required to be documented.

Common Implementations

Many vendors include an appendix in their documentation where all implementation-defined behavior is collected together.

Of necessity a vendor will need to document extensions if their customers are to make use of them. Whether they document all extensions is another matter. One method of phasing out a superseded extension is to cease documenting it, but to continue to support it in the implementation. This enables programs that use the extension to continue being translated, but developers new to that implementation will be unlikely to make use of the extension (not having any documentation describing it).

Coding Guidelines

For those cases where use of implementation-defined behavior is being considered, the vendor implementation-provided document will obviously need to be read. The commercially available compiler validation suites do not check implementation-defined behavior. It is recommended that small test programs be written to verify that an implementation's behavior is as documented.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types <float.h> (7.7), alternative spellings <iso646.h> (7.9), sizes of integer types <limits.h> (7.10), variable arguments <stdarg.h> (7.15), boolean type and values <stdbool.h> (7.16), common definitions <stddef.h> (7.17), integer types <stdint.h> (7.18). 101

---

footnote 4
4) Strictly conforming programs are intended to be maximally portable among conforming implementations. 102

Commentary

strictly conforming program
use features of language/library
A strictly conforming program is acceptable to all conforming implementations.

C++

The word *portable* does not occur in the C++ Standard. This may be a consequence of the conformance model which is based on implementations, not programs.

Example

It is possible for a strictly conforming program to produce different output with different implementations, or even every time it is compiled:

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
printf("INT_MAX=%d\n", INT_MAX);
printf("Translated date is %s\n", __DATE__);
}
```

---

103 Conforming programs may depend upon nonportable features of a conforming implementation.

conforming programs may depend on

Commentary

What might such *nonportable* features be? The standard does not specify any construct as being nonportable. The only other instance of this term occurs in the definition of undefined behavior. One commonly used meaning of the term *nonportable* is a construct that is not likely to be available in another vendor's implementation. For instance, support for some form of inline assembler code is available in many implementations. Use of such a construct might not be considered as a significant portability issue.

46 undefined behavior

C++

While a conforming implementation of C++ may have extensions, 1.4p8, the C++ conformance model does not deal with programs.

Coding Guidelines

There are a wide range of constructs and environment assumptions that a program can make to render it nonportable. Many nonportable constructs tend to fall into the category of undefined and implementation-defined behaviors. Avoiding these could be viewed, in some cases, as being the same as avoiding nonportable constructs.

Example

Relying on `INT_MAX` being larger than 32,767 is a dependence on a nonportable feature of a conforming implementation.

```
#include <limits.h>

_Bool f(void)
{
return (32767 < INT_MAX);
}
```

5. Environment

---

104 An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this International Standard.

environment execution

Commentary

For a hosted implementation the two environments are often the same. In some cases application developers do cross-translate from one hosted environment to another hosted environment. In a freestanding environment,

93 conforming hosted implementation
155 freestanding environment startup

the two environments are very unlikely to be the same.

A commonly used term for the execution environment is *runtime system*. In some cases this terminology refers to a more restricted set of functionality than a complete execution environment.

diagnostic 146 shall produce

The requirement on when a diagnostic message must be produced prevents a program from being translated from the source code, on the fly, as statements to execute are encountered.

Rationale

Because C has seen widespread use as a cross-compiled cross-compilation language, a clear distinction must be made between translation and execution environments. The C89 preprocessor, for instance, is permitted to evaluate the expression in a `#if` directive using the long integer or unsigned long integer arithmetic native to the translation environment: these integers must comprise at least 32 bits, but need not match the number of bits in the execution environment. In C99, this arithmetic must be done in `intmax_t` or `uintmax_t`, which must comprise at least 64 bits and must match the execution environment. Other translation time arithmetic, however, such as type casting and floating point arithmetic, must more closely model the execution environment regardless of translation environment.

C++

The C++ Standard says nothing about the environment in which C++ programs are translated.

Other Languages

JIT

Java defines an execution environment. It says nothing about the translation environment. But its philosophy, *write once run anywhere* means that there should not be any implementation-defined characteristics to worry about. There are implementations that perform Just-in-time (JIT) translation on an as needed basis during execution (implementations differ in the granularity of source that is JIT translated).

Coding Guidelines

Coding guidelines often relate to the translation environment; that is, what appears in the visible source code. In some cases the behavior of a program may vary because of characteristics that only become known when a program is executed. The coding guidelines in this book are aimed at both environments. It is management's responsibility to select the ones (or remove the ones) appropriate to their development environment.

Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations. 105

Commentary

The translation environment need not have any effect on the translated program, subject to sufficient memory being available to perform a translation. It is not even necessary that the translation environment be a superset of the execution environment. For instance, a translator targeting a 64-bit execution environment, but running in a 32-bit translation environment, could support its own 64-bit arithmetic package (for constant folding).

translation phase 4 129

In theory each stage of translation could be carried out in a separate translation environment. In some development environments, the code is distributed in preprocessed (i.e., after translation phase 4) form. Header files will have been included and any conditional compilation directives executed.

In those cases where a translator performs operations defined to occur during program execution, it must follow the execution time behavior. For instance, a translator may be able to evaluate parts of an expression, that are not defined to be a constant expression. In this case any undefined behavior associated with a signed arithmetic overflow could be defined to be the diagnostic generated by the translator.

C++

The C++ Standard makes no such observation.

Coding Guidelines

The characteristics of the execution environment are usually thought of as being part of the requirements of the application (i.e., that the application is capable of execution in this environment). The characteristics of the translation environment are of interest to these coding guidelines if they may affect the behavior of a translator.

106 **Forward references:** In this clause, only a few of many possible forward references have been noted.

Commentary

This statement could be said to be true for all of the Forward references appearing in the C Standard.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

107 A C program need not all be translated at the same time.

program not translated at same time

Commentary

C's separate compilation model is one of independently translated source files that are merged together by a linker to form a program image. There is no concept of program library built into the language. Neither is there any requirement to perform cross-translation unit checking, although there are cross-translation unit compatibility rules for derived types.

1810 translation unit syntax
113 translation unit linked
141 program image
633 compatible separate translation units

There is no requirement that all source files making up a C program be translated prior to invoking the function `main`. An implementation could perform a JIT translation of each source file when an object or function in an untranslated source file is first referenced (a translator is required to issue a diagnostic if a translation unit contains any syntax and constraint violations).

104 JIT

Linkage is the property used to associate the same identifier, declared in different translation units, with the same object or function.

420 linkage

Other Languages

Some languages enforce strict dependency and type checks between separately translated source files. Others have a very laid-back approach. Some execution environments for the Basic language delay translation of a declaration or statement until it is reached in the flow of control during program execution. A few languages require that a program be completely translated at the same time (Cobol and the original Pascal standard).

Java defines a process called resolution which, " . . . is optional at the time of initial linkage."; and "An implementation may instead choose to resolve a symbolic reference only when it is actively used; . . . ".

Common Implementations

Most implementations translate individual source files into object code files, sometimes also called object modules. To create a program image, most implementations require all referenced identifiers to be defined and externally visible in one of these object files.

Coding Guidelines

The C model could be described as one of *it's up to you to build it correctly or the behavior is undefined.*. Having all of the source code of a program in a single file represents poor practice for all but the smallest of programs. The issue of how to divide up source code into different sources files, and how to select what definitions go in what files, is discussed elsewhere. There is also a guideline recommendation dealing with the uniqueness and visibility of declarations that appear at file scope.

1810 external declaration syntax
422.1 identifier declared in one file

Example

The following is an example of objects declared in different translation units with different types.

file_1.c

```
1  extern int glob;
```

file_2.c

```
1  float glob = 1.0;
```

Usage

A study by Linton and Quong[858] used an instrumented `make` program to investigate the characteristics of programs (written in a variety of languages, including C) built over a six-month period at Stanford University. The results (see Figure 107.1) showed that approximately 40% of programs consisted of three or fewer translation units.

---

source files
preprocessing files

The text of the program is kept in units called *source files*, (or *preprocessing files*) in this International Standard. 108

Commentary

This defines the terms *source files* and *preprocessing files*. The term *source files* is commonly used by developers, while the term *preprocessing files* is an invention of the Committee.

C90

The term *preprocessing files* is new in C99.

C++

The C++ Standard follows the wording in C90 and does not define the term *preprocessing files*.

Other Languages

The Java language specification[508] strongly recommends that certain naming conventions be followed for package names and class files. The names mimic the form of Web addresses and RFCs 920 and 1032 are cited.

Common Implementations

A well-established convention is to suffix source files that contain the object and function definitions with the `.c` extension. Header files usually being given a `.h` suffix. This convention is encoded in the `make` tool, which has default rules for processing file names that end in `.c`.

Coding Guidelines

file name abbreviations

Restrictions on the number of characters in a filename are usually more severe than for identifiers (MS-DOS 8.3, POSIX 14). These restrictions can lead to the use of abbreviations in the naming of files. An automated tool developed by Anquetil and Lethbridge[48] was able to extract abbreviations from file names with better than 85% accuracy. A comparison of automated file clustering,[49] against the clustering of files in a large application, by a human expert, showed nearly 90% accuracy for both precision (files grouped into subsystems to which they do not belong) and recall (files grouped into subsystems to which they do belong).

Development groups often adopt naming conventions for source file names. Source files associated with implementing particular functionality have related names, for instance:

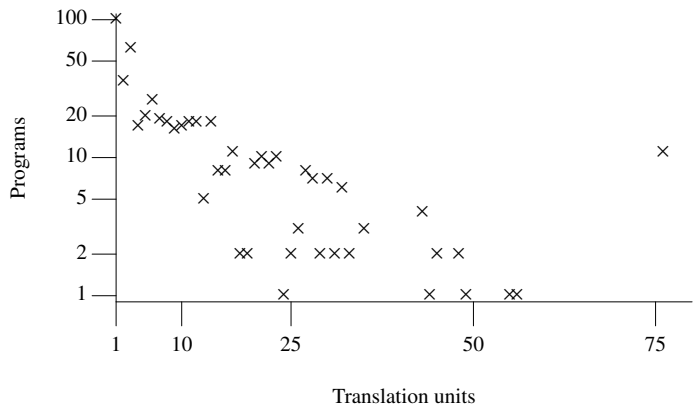


Figure 107.1: Number of programs built from a given number of translation units. Adapted from Linton.[858]

1. Data manipulation: *db* (database), *str* (string), or *queue*.
2. Algorithms or processes performed: *mon* (monitor), *write*, *free*, *select*, *cnv* (conversion), or *chk* (checking).
3. Program control implemented: *svr* (server), or *mgr* (manager).
4. The time period during which processing occurs: *boot*, *ini* (initialization), *rt* (runtime), *pre* (before some other task), or *post* (after some other task).
5. I/O devices, services or external systems interacted with: *k2*, *sx2000*, (a particular product), *sw* (switch), *f* (fiber), *alarm*.
6. Features implemented: *abrvdial* (abbreviated dialing), *mtce* (maintenance), or *edit* (editor).
7. Names of other applications from where code has been reused.
8. Names of companies, departments, groups or individuals who developed the code.
9. Versions of the files or software (e.g., the number 2 or the word *new* may be added, or the name of target hardware), different versions of a product sold in different countries (e.g., *na* for North America, and *ma* for Malaysia).
10. Miscellaneous abbreviations, for instance: *utl* (utilities), or *lib* (library).

792 identifier selecting spelling

The standard has no concept of directory structure. The majority of hosts support a file system having a directory structure and larger, multisource file projects often store related source files within individual directories. In some cases the source file directory structure may be similar to the structure of the major components of the program, or the directory structure mirrors the layered structure of an application.[790] The issues involved in organizing names into the appropriate hierarchy are discussed later.

530 structure type sequentially allocated objects

Files are not the only entities having names that can be collected into related groups. The issues associated with naming conventions, the selection of appropriate names and the use of abbreviations. are discussed elsewhere.

517 enumeration set of named constants
792 abbreviating identifier
792 identifier introduction

Source files are not the only kind of file discussed by the C Standard. The **#include** preprocessing directive causes the contents of a file to be included at that point. The standard specifies a minimum set of requirements for mapping these header files. The coding guideline issues associated with the names used for these headers is discussed elsewhere.

1896 source file inclusion
422 header name same as .c file

---

109 A source file together with all the headers and source files included via the preprocessing directive **#include** is known as a *preprocessing translation unit*.

preprocessing translation unit known as

Commentary

This defines the term *preprocessing translation unit*, which is not generally used outside of the C Standard Committee. A preprocessing translation unit contains all of the possible combinations of translation units that could appear after preprocessing. A preprocessing translation unit is parsed according to the syntax for preprocessing directives.

1854 preprocessor directives syntax

C90

The term *preprocessing translation unit* is new in C99.

C++

Like C90, the C++ Standard does not define the term *preprocessing translation unit*.

Other Languages

Java defines the term *compilation unit*. Other terms used by languages include *module*, *program unit*, and *package*.

Coding Guidelines

Use of this term by developers is almost unknown. The term *source file* is usually taken to mean a single file, not including the contents of any files that may be **#include**d. Although a slightly long-winded term, *preprocessing translation unit* is the technically correct one. As such its use should be preferred in coding guideline documents.

---

translation unit known as

After preprocessing, a preprocessing translation unit is called a *translation unit.* 110

Commentary

This defines the term *translation unit*. A translation unit is the sequence of tokens that are the output of translation phase 4. The syntax for translation units is given elsewhere.

translation phase 4 129
translation unit syntax 1810

C90

A source file together with all the headers and source files included via the preprocessing directive ***#include****, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a translation unit.*

This definition differs from C99 in that it does not specify whether macro definitions are part of a translation unit.

C++

The C++ Standard, 2p1, contains the same wording as C90.

Common Implementations

In many translators the task of turning a preprocessing translation unit into a translation unit is the job of a single program .

footnote 5 120

Coding Guidelines

Although the term *translation unit* is defined by the standard to refer to the sequence of tokens *after preprocessing*; the term is not commonly used by developers. The term *after preprocessing* is commonly used by developers to refer to what the standard calls a translation unit. There seems to be little to be gained in trying to change this common usage term.

translation phase 7 136

Some of these coding guidelines apply to the sequence of tokens input to translation phase 7 (semantic analysis). There is rarely any difference between what goes into phase 5 and what goes into phase 7, and the term *after preprocessing* is commonly used by developers. For simplicity all phrases of translation after preprocessing are lumped together as a single whole. The guidelines in this book apply either to the visible source, before preprocessing, or after preprocessing.

---

translation units preserved

Previously translated translation units may be preserved individually or in libraries. 111

Commentary

The standard does not specify what information is preserved in these translated translation units. It could be a high-level representation, even some tokenized form, of the original source. It is most commonly relocatable machine code and an associated symbol table.

The standard says nothing about the properties of libraries, except what is stated here.

Other Languages

Some languages specify the information available from, or the properties of, their separately translated translation units. Many languages, for instance Fortran, do not even specify as much as the C Standard.

Java defines several requirements for how packages are to be preserved and suggests several possibilities; for instance, class files in a hierarchical file system with a specified naming convention.

Common Implementations

In the Microsoft Windows environment, translated files are usually given the suffix `.obj` and libraries the suffix `.lib` or `.dll`. In a Unix environment, the suffix `.o` is used for object files and the suffix `.a` for library files, or `.so` for dynamically linked libraries.

Coding Guidelines

Coding guidelines, on the whole, do not apply to the translated output. Use of tools, such as `make`, for ensuring consistency between libraries and the translated translation unit they were built from, and the source code that they were built from, are outside the scope of this book.

---

112 The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files.

translation units communication between

Commentary

Translation phase 8 is responsible for ensuring that all references to external objects and functions refer to the same entity. The technical details of how an object or function referenced in one translation unit accesses the appropriate definition in another translation unit is a level of detail that the standard leaves to the implementation. The issues of the types of these objects agreeing with each other, or not, is discussed elsewhere.

139 translation phase 8
1810 translation unit syntax
633 compatible separate translation units

Data read from a binary file is always guaranteed to compare equal to the data that was earlier written to the same file, during the same execution of a program. Separate translation units can also communicate by manipulating objects through pointers to those objects. These objects are not restricted to having external linkage. Similarly, functions can also be called via pointers to them. Visible identifiers denoting object or function definitions are not necessary.

Common Implementations

Information on the source file in which a particular function or object was defined is not usually available to the executing program. However, hosts that support dynamic linking provide a mechanism for implementations to locate functions that are referenced during program execution (most implementations require objects to have storage allocated to them during program startup).

150 program startup

Coding Guidelines

The issue of deciding which translation unit should contain which definition is discussed elsewhere, as is the issue of keeping identifiers declared in different translation units synchronized with each other.

1810 declarations in which source file
422.1 identifier declared in one file

---

113 Translation units may be separately translated and then later linked to produce an executable program.

translation unit linked

Commentary

This is all there is to the C model of separate compilation. The C Standard places no requirements on the linking process, other than producing a program image. How the translation units making up a complete program are identified is not specified by the standard. The input to translation phase 8 requires, under a hosted implementation, at least a translation unit that contains a function called `main` to create a program image.

107 program not translated at same time
162 hosted environment startup

Common Implementations

Most translators have an option that specifies whether the source file being translated should be linked to produce a program image (translation phase 8), or the output from the translator should be written to an object file (with no linking performed). In a Unix environment, the convention is for the default name of the file containing the executable program to be `a.out`.

---

114 **Forward references:** linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

translation phases of

The precedence among the syntax rules of translation is specified by the following phases.[5)]

Commentary

These phases were introduced by the C committee to answer translation ordering issues that differed among early C implementations.

If one or more source files is **#include**d, the phases are applied, in sequence, to each file. So it is not possible for constructs created prior to phase 4 (which handles **#include**) to span more than one source file. For instance, it is not possible to open a comment in one file and close it in another file. Constructs that occur after phase 4 can span multiple files. For instance, a string literal as the last token in one file can be concatenated to a string literal which is the first token in an immediately **#include**d file.

universal character name syntax 815

The following quote from the Rationale does not belong within any specific phrase of translation, so it is provided here. UCNs are discussed elsewhere.

UCN models of

Rationale

> . . . , how to specify UCNs in the Standard. Both the C and C++ Committees studied this situation and the available solutions, and drafted three models:
>
> A. Convert everything to UCNs in basic source characters as soon as possible, that is, in translation phase 1.
>
> B. Use native encodings where possible, UCNs otherwise.
>
> C. Convert everything to wide characters as soon as possible using an internal encoding that encompasses the entire source character set and all UCNs.
>
> Furthermore, in any place where a program could tell which model was being used, the standard should try to label those corner cases as undefined behavior.

C++

C++ has nine translation phases. An extra phase has been inserted between what are called phases 7 and 8 in C. This additional phase is needed to handle templates, which are not supported in C. The C++ Standard specifies what the C Rationale calls model A.

C++ model A 116

Other Languages

Most languages do not contain a preprocessor, and do not need to go to the trouble of explicitly calling out phases of translation. The C Standard might not have had to do this either, had it not been for the differing interpretations of the base document made by some translators.

base document 1

Java has three lexical translation steps (the first two are needed to handle Unicode).

Coding Guidelines

Few developers have any detailed knowledge of the phases of translation. It can be argued that use of constructs whose understanding requires a detailed knowledge of the phases of translation should be avoided. The problem is how to quantify *detailed knowledge*. Coding guidelines apply in all phases of translation, unless stated otherwise.

The distinction between preprocessor and subsequent phases is a reasonably well-known and understood division. The processes used by developers for extracting information from source code is likely to be affected by their knowledge of how a translator operates. Thinking in terms of the full eight phases is often unnecessary and overly complicated. The following phases are more representative of how developers view the translation process:

1. Preprocessing.
2. Syntax, semantics, and machine code generation.
3. Linking.

Example

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  printf("\\" "101");
6  }
```

---

116 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. translation phase 1

Commentary

This phase maps the bits held on some storage device onto members of the source character set (which is defined elsewhere). C requires that sequences of source file characters be grouped into units called lines (actually there are two kinds of lines). There is a lot of variability between hosts on how end-of-line is indicated. This specification requires that whatever method is used at the physical level, it be mapped to a single new-line indicator. 214 source character set 118 translation phase 2 224 end-of-line representation

The source file being translated may reside on host A, with the implementation doing the translation may be executing on host B, and the translated program may be intended to run on host C. All three hosts could be using different character set representations. During this phase of translation, we are only interested in host A and host B. The character set used by host C is of no consequence, to the translator, until translation phase 5. 133 translation phase 5

C90

In C90 the source file contains characters (the 8-bit kind), not multibyte characters.

C++

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. . . . Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character. 2.1p1

```
1  #define mkstr(s) #s
2
3  char *dollar = mkstr($); // The string "\u0024" is assigned
4                           /* The string "$", if that character is supported */
```

C++ model A

The C++ Committee defined its Standard in terms of model A, just because that was the clearest to specify (used the fewest hypothetical constructs) because the basic source character set is a well-defined finite set. Rationale

The situation is not the same for C given the already existing text for the standard, which allows multibyte characters to appear almost anywhere (the most notable exception being in identifiers), and given the more low-level (or *close to the metal*) nature of some uses of the language.

Therefore, the C committee agreed in general that model B, keeping UCNs and native characters until as late as possible, is more in the “spirit of C” and, while probably more difficult to specify, is more able to encompass

the existing diversity. The advantage of model B is also that it might encompass more programs and users' intents than the two others, particularly if shift states are significant in the source text as is often the case in East Asia.

In any case, translation phase 1 begins with an implementation-defined mapping; and such mapping can choose to implement model A or C (but the implementation must document it). As a by-product, a strictly conforming program cannot rely on the specifics handled differently by the three models: examples of non-strict conformance include handling of shift states inside strings and calls like `fopen("\\ubeda\\file.txt","r")` and `#include "sys\udefault.h"`. Shift states are guaranteed to be handled correctly, however, as long as the implementation performs no mapping at the beginning of phase 1; and the two specific examples given above can be made much more portable by rewriting these as `fopen("\\" "ubeda\\file.txt", "r")` and `#include "sys/udefault.h"`.

Other Languages

Java 3.2 *A translation of the Unicode escapes (3.3) in the raw stream of Unicode characters to the corresponding Unicode character . . .*

ISO 646 24 Which means that characters other than those appearing in ISO/IEC 646 can appear in identifiers, strings and character constants, etc.

Common Implementations

This phase is where a translator interfaces with the host to read sequences of bytes from a source file. A source file is usually represented as a text file. There are a few hosts that have no concept of a text file. Some translators use relatively high-level system routines and rely on the host to return a line of characters; others perform block reads of binary data and perform their own character mappings. The choice will depend on the facilities offered by the host and the extent to which a translator wants to get involved in unpicking the format used to store characters in a source file. Some hosts treat text files (the usual method for storing source files) differently from binary files (lines are terminated and end-of-file may be indicated by a special character or trailing null characters).

There is no requirement that the file containing C source code have any particular form. Known forms include the following:

- *Stream of bytes.* Both text and binary files are treated as a linear sequence of bytes— the Unix model.
- *Text files have special end-of-line markers and end-of-file is indicated by a special character.* Binary files are treated as a sequence of bytes.
- *Fixed-length records.* These records can be either fixed-line length (a line cannot contain more than a given, usually 72 or 80, number of characters; dating back to when punch cards were the primary form of input to computers), or fixed-block length (i.e., lines do not extend over block boundaries and null characters are used to pad the last line in a block).

A translator that reads a block of characters at a time has to be responsible for knowing the representation of source files and may, or may not, have to perform some conversion to create an end-of-line indicator.[447]

Source files are usually represented in storage using the same set of byte values that are used by the translator to represent the source character set, so there is no actual mapping involved in many cases. The physical representation used to represent source files will be chosen by the tools used to create the source file, usually an editor.

The Unisys A Series[1390] uses fixed-length records. Each record contains 72 characters and is padded on the right with spaces (no new-line character is stored). To represent logical lines that are longer than 72

characters, a backslash is placed in column 72 of the physical line, folding characters after the 71 onto the next physical line. A logical line that does end in a backslash character is represented in the physical line by two backslash characters.

The Digital Mars C[356] compiler performs special processing if the input file name ends in `.htm` or `.html`. In this case only those characters bracketed between the HTML tags <code> and </code> are considered significant. All other characters in the input file are ignored.

The IBM ILE C development environment[617] associates a Coded Character Set Identifier (CCSID) with a source physical file. This identifier denotes the encoding used, the character set identifiers, and other information. Files that are **#include**d may have different CCSID values. A set of rules is defined for how the contents of these include files is mapped in relation to CCSID of the source files that **#include**d them. A **#pragma** preprocessing directive is provided to switch between CCSIDs within a single source file; for instance:

```
1  char EBCDIC_hello[] = "Hello World";
2
3  /* Switch to ASCII character set. */
4  #pragma convert(850)
5  char ASCII_hello[] = "Hello World";
6
7  /* Switch back. */
8  #pragma convert(0)
```

Example

WG14/N770

If the source contains:

$??)

and the translator is operating in a locale where $ and the immediately following character represent a single multibyte character. Then the input stream consists of the multibyte characters: $? ?)

In another locale the input stream might consist of the multibyte characters: $? ?) with the ??) being treated as a trigraph sequence and replaced by].

Table 116.1: Total number of characters and new-lines in the visible form of the `.c` and `.h` files.

	`.c` files	`.h` files
total characters	192,165,594	64,429,463
total new-lines	6,976,266	1,811,790
non-comment characters	144,568,262	43,485,916
non-comment new-lines	6,113,075	1,491,192

117 Trigraph sequences are replaced by corresponding single-character internal representations.

trigraph sequences phase 1

Commentary

The replacement of trigraphs by their corresponding single-character occurs before preprocessing tokens are created. This means that the replacement happens for all character sequences, not just those outside of string literals and character constants.

233 trigraph sequences mappings
895 string literal syntax
867 integer character constant

Other Languages

Many languages are designed with an Ascii character set in mind, or do not contain a sufficient number of punctuators and operators that all characters not in a commonly available subset need to be used. Pascal specifies what it calls lexical alternatives for some lexical tokens.

Common Implementations

Studies of translator performance have shown that a significant amount of time is consumed by lexing characters to form preprocessing tokens.[1436] In order to improve performance for the average case (trigraphs are not frequently used), one vendor (Borland) wrote a special program to handle trigraphs. A source file that contained trigraphs first had to be processed by this program; the resulting output file was then fed into the program that implemented the rest of the translator.

Coding Guidelines

Because the replacement occurs in translation phase 1, trigraphs can have unexpected effects in string literals and character constants. Banning the use of trigraphs will not prevent a translator from replacing them if encountered in the source. Also, in string literal contexts the developers mind-set is probably not thinking of trigraphs, so such sequences are unlikely to be noticed anyway.

Sequences of ? characters may be needed within literals by the application. One solution is to replace the second of the ? characters by the escape sequence \?, unless a trigraph is what was intended.

Some guidelines suggest running translators in a nonstandard mode (some translators provide an option that causes trigraph sequences to be left unreplaced), if one exists, as a way of preventing trigraph replacement from occurring. Running a translator in a nonstandard mode is rarely a good idea; what of those developers who are aware of trigraphs and intentionally use them?

The use of trigraphs may overcome the problem of entering certain characters on keyboards. But visually they are not easily processed, or to be exact very few developers get sufficient practice reading trigraphs to be able to recognize them effortlessly. Digraphs were intended as a more readable alternative (the characters used are more effective memory prompts for recalling the actual character they represent; they are discussed elsewhere).

digraphs 916

Example

```
#include <stdio.h>

void f(void)
{
printf("??=");     /* Prints # */
printf("?" "?=");  /* Prints ??= */
printf("?\?=");    /* Prints ??= */
}
```

Usage

The visible form of the `.c` files contain 8 trigraphs (`.h` 0).

---

translation phase 2
physical source line
logical source line
line splicing

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. 118

Commentary

This process is commonly known as *line splicing*. The preprocessor grammar requires that a directive exists on a single logical source line. The purpose of this rule is to allow multiple physical source lines to be spliced to form a single logical source line so that preprocessor directives can span more than one line. Prior to the introduction of string concatenation, in C90, this functionality was also used to create string literals that may have been longer than the physical line length, or could not be displayed easily by an editor.

Emailing source code is now common. Some email programs limit the number of characters on a line and will insert line breaks if this limit is exceeded. Human-written source might not form very long lines, but automatically generated source can sometimes contain very long identifier names.

C++

The first sentence of 2.1p2 is the same as C90.

The following sentence is not in the C Standard: 2.1p2

If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined.

```
#include <stdio.h>

int \u1F\
5F;          // undefined behavior
             /* defined behavior */
void f(void)
{
printf("\\u0123"); /* No UCNs. */
printf("\\u\
0123"); /* same as above, no UCNs */
        // undefined, character sequence that matches a UCN created
}
```

Common Implementations

Some implementations use a fixed-length buffer to store logical source lines. This does not necessarily imply that there is a fixed limit on the maximum number of characters on a line. But encountering a line longer than the input buffer can complicate the generation of log files and displaying the input line with any associated diagnostics. Both quality-of-implementation issues are outside the scope of the standard.

Coding Guidelines

A white-space character is sometimes accidentally placed after a backslash. This can occur when source files are ported, unconverted between environments that use different end-of-line conventions; for instance, reading MS-DOS files under Linux. The effect is to prevent line splicing from occurring and invariably causes a translator diagnostic to be issued (often syntax-related). This is an instance of unintended behavior and no guideline recommendation is made.

The limit on the number of characters on a logical source line is very unlikely to be reached in practice and line splicing is rarely used outside of preprocessing directives. Existing source sometimes uses line splicing to create a string literal spanning more than one source code line. The reason for this usage often is originally based on having to use a translator that did not support string literal concatenation.

292 limit characters on line

Example

```
#include <stdio.h>

#define X          ??/
           (1+1) /* ??/ -> \ */

extern int g\
l\
o\
b
;

void f(void)
{
if (glob)
   {
   printf("Something so verbose we need\
 to split it over more than one line\n");
   printf ("Something equally verbose but at"
```

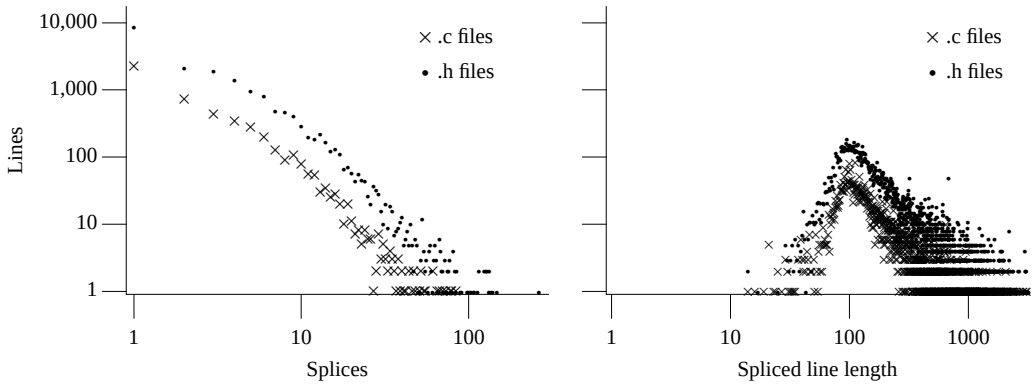


Figure 118.1: Number of physical lines spliced together to form one logical line and the number of logical lines, of a given length, after splicing. Based on the visible form of the `.c` and `.h` files.

```
            " least we have some semblance of visual layout\n");
    }

printf("\\u0123"); /* No UCNs. */
printf("\\
\u0123");          /* Same as above, no UCNs. */

printf("\\
\n");              /* No new-line output. */
}
```

Usage

In the visible form of the `.c` files 0.21% (`.h` 4.7%) of all physical lines are spliced. Of these line splices 33% (`.h` 7.8%) did not occur within preprocessing directives (mostly in string literals).

Only the last backslash on any physical source line shall be eligible for being part of such a splice. 119

Commentary

A series of backslash characters at the end of a line does not get consumed (assuming there are sufficient empty following lines). This is a requirement that causes no code to be written in the translator, as opposed to a requirement that needs code to be written to implement it.

C90

This fact was not explicitly specified in the C90 Standard.

C++

The C++ Standard uses the wording from C90.

Example

```
#include <stdio.h>

void f(void)
{
/*
 * In the following the two backslash characters do not cause two
 * line splices.  There is a single line splice.  This results in
 * a single double-quote character, causing undefined behavior.
 */
printf("\\
```

```
11
12  n");
13  /*
14   * The above is not equivalent to printf("n");
15   */
16
17  /*
18   * Below the backslash characters are on different lines.
19   */
20  printf("\
21  \
22  n");
23  }
```

120 5) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice. footnote 5

Commentary

The phases of translation describe an intended effect, not an implementation strategy. It is not expected that implementations implement the different phases via different programs.

Common Implementations

Many translators do split up the job of translation between a number of programs. Typically one program performs preprocessing (translation phases 1–4), while another performs syntax, semantic analysis, and code generation (the last operation is not directly mentioned in the standard). The usual method of communicating between the two programs is via intermediate files. If the original source file has the name `f.c` and translator options are used to save the output of various translation phases, the file holding the preprocessed output is normally given the name `f.i` and the file holding any generated assembler code is given the name `f.s`. Phase 8 is nearly always performed by a separate program (that can usually also handle languages other than C), a linker. [0] translation technology

A compiler sold by Borland included a separate program to handle trigraphs (the programs handling other phases of translation did not include code to process trigraphs).

At least one program, `lcc`,[448] effectively only performs phase 7. It requires a third-party program to perform the earlier and later phases. The method of communication between phases is a file containing a sequence of characters that look remarkably like a preprocessed file (so `lcc` has to retokenize its input).

source file representation

121 Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation.

Commentary

The term *file* has a common usage within computing and the term *source file* could be interpreted to imply that source code had to be stored in such files. While source files are commonly represented using a text file within a host file system there is no requirement to use such a representation.

A translator may chose to internally maintain information about the effect of including a system header (e.g., an internal symbol table of declared identifiers) that is accessed when the corresponding **`#include`** is encountered. In such an implementation there is no external representation of the system header.

This sentence was added by the response to DR #308.

Other Languages

Languages which are defined by a written specification do not usually require that a particular external representation be used for source files. Languages defined by a particular implementation (e.g., PERL) require a source file representation that can be handled by that implementation.

Common Implementations

header precompiled

Some implementations support what are known as *precompiled headers*.[755, 860] The contents of such headers have a form that has been partially processed through some phases of translation. The benefit of using precompiled headers is a, sometimes dramatic, improvement in rate of translation (figures of 20–70% have been reported).

IDE

Some software development environments (often called *IDE*s', Integrated Development Environments) hold the source code within some form of database. This database often includes version-control information, translator options, and other support information.

The description is conceptual only, and does not specify any particular implementation. 122

Commentary

as-if rule

The term *as-if rule* (or sometimes *as-if principle*) occurs frequently in discussions involving the C Standard. This term is not defined in the C Standard, but is mentioned in the Rationale:

Rationale

> The as if principle is invoked repeatedly in this Rationale. The C89 Committee found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft the models so that implementors are constrained only insofar as they must bring about the same result, as if they had implemented the presentation model; often enough the clearest model would make for the worst implementation.
>
> A question sometimes asked regarding optimization is, "Is the rearrangement still conforming if the precomputed expression might raise a signal (such as division by zero)?" Fortunately for optimizers, the answer is "Yes," because any evaluation that raises a computational signal has fallen into an undefined behavior (§6.5), for which any action is allowable.

Essentially, a translator is free to do what it likes as long as the final program behaves, in terms of visible output and effects, as-if the semantics of the abstract machine were being followed. In some instances the standard calls out cases based on the as-if rule.

expression 190 need not evaluate part of

This sentence was added by the response to DR #308.

C++

1.9p1

> *In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.5)*

Footnote 5

> *This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.*

---

source file end in new-line

A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place. 123

Commentary

What should the behavior be if the last line of an included file did not end in a new-line? Should the characters at the start of the line following the **#include** directive be considered to be part of any preceding preprocessing token (from the last line of the included file)? Or perhaps source files should be treated as containing an implicit new-line at their end. This requirement simplifies the situation by rendering the behavior undefined.

125 source file partial preprocessing token

Lines are important in preprocessor directives, although they are not important after translation phase 4. Treating two apparently separate lines, in two different source files, as a single line opens the door to a great deal of confusion for little utility.

C90

The wording, " . . . before any such splicing takes place.", is new in C99.

Coding Guidelines

While undefined behavior will occur for this usage, instances of it occurring are so rare that it is not worth creating a coding guideline recommending against its use.

Example

```
1  /*
2   * If this source file is #include'd by another source file, might
3   * some implementation splice its first line onto the last line?
4   */
5  void f(void)
6  {
7  }\
```

---

124 3. The source file is decomposed into preprocessing tokens[6)] and sequences of white-space characters (including comments). translation phase 3

Commentary

Preprocessing tokens are created before any macro substitutions take place. The C preprocessor is thus a token preprocessor, not a character preprocessor. The base document was not clear on this subject and some implementors interpreted it as defining a character preprocessor. The difference can be seen in:

925 EXAMPLE tokenization
1 base document

```
1  #define a(b) printf("b=%d\n", b);
2
3  a(var);
```

The C preprocessor expands the above to:

```
1  printf("b=%d\n", var);
```

while a character preprocessor would expand it to:

```
1  printf("var=%d\n", var);
```

Linguists used the term *lexical analysis* to describe the process of collecting characters to form a word before computers were invented. This term is used to describe the process of building preprocessing tokens and in C's case would normally be thought to include translation phases 1–3. The part of the translator that performs this role is usually called a *lexer*. As well as the term *lexing*, the term *tokenizing* is also used.

Common Implementations

Decomposing a source file into preprocessing tokens is straight-forward when starting from the first character. However, in order to provide a responsive interface to developers, integrated development environments often perform incremental lexical analysis[1433] (e.g., only performing lexical analysis on those characters in the source that have changed, or characters that are affected by the change).

Coding Guidelines

The term *preprocessing token* is rarely used by developers. The term *token* is often used generically to apply to such entities in all phases of translation.

Usage

The visible form of the `.c` files contain 30,901,028 (`.h` 8,338,968) preprocessing tokens (new-line not included); 531,677 (`.h` 248,877) `/* */` comments, and 52,531 (`.h` 27,393) `//` comments.

preprocessing tokens 777
white space separation

Usage information on white space is given elsewhere.

---

source file partial preprocessing token

A source file shall not end in a partial preprocessing token or in a partial comment. 125

Commentary

What is a partial preprocessing token? Presumably it is a sequence of characters that do not form a preprocessing token unless additional characters are appended. However, it is always possible for the individual characters of a multiple-character preprocessing token to be interpreted as some other preprocessing token (at worst the category "each non-white-space character that cannot be one of the above" applies). For instance, the two characters `..` (where an additional period character is needed to create an ellipsis preprocessing token) represents two separate preprocessing tokens (e.g., two periods). The character sequence `%:%` represents the two preprocessing tokens `#` and `%` (rather than `##`, had a `:` followed).

preprocessing token 770
syntax

The intent is to make it possible to be able perform low-level lexical processing on a per source file basis. That is, an **`#include`**d file can be lexically analyzed separately from the file from which it was included. This means that developers only need to look at a single source file to know what preprocessing tokens it contains. It can also simplify the implementation.

source file 123
end in new-line

The requirement that source files end in a new-line character means that the behavior is undefined if a line (physical or logical) starts in one source file and is continued into another source file.

In this phase a comment is an indivisible unit. A source file cannot contain part of such a unit, only a whole comment. That is, it is not possible to start a comment in one source file and end it in another source file.

Coding Guidelines

Translators are not required to diagnose a comment that starts in one source file and ends in another. However, this usage is very rare and consequently a guideline recommendation is not cost effective.

---

comment replaced by space

Each comment is replaced by one space character. 126

Commentary

The C committee had to choose how many space characters a comment was converted into, including zero. The zero case had the disadvantage of causing some surprising effects, although some existing implementations had gone down this route. They finally decided that specifying more than a single space character was of dubious utility. Replacing a comment by one space character minimizes the interaction between it and the adjacent preprocessing tokens (space characters are not usually significant).

white space 780
significant

Other Languages

Java supports the `/* */` form of comments. The specification does not say what they get converted into.

Common Implementations

base document 1

The base document replaced a comment by nothing (some implementations continue to support this functionality for compatibility with existing code[600, 1314]). This had the effect of treating:

```
1  int a/* comment */b;
```

as a declaration of the identifier ab. The C Committee introduced the ## operator to explicitly provide this functionality. 1958 ## operator

Example

```
1  #define mkstr(a) #a
2
3  char *p = mkstr(x/* a comment*/y); /* p points at the string literal "x y" */
```

---

127 New-line characters are retained.

Commentary

New-line is a token in the preprocessor grammar. It is used to delimit the end of preprocessor directives.

Other Languages

New-line is important in several languages. Some older languages, and a few modern ones, have given meaning to new-line. Fortran (prior to Fortran 90) was not free format; the end of a line is the end of a statement or declaration, unless a line-continuation character appears in column 5 of the following line.

---

128 Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

white-space sequence replaced by one

Commentary

In this phase the only remaining white-space characters that need to be considered are those that occur between preprocessing tokens. All other white-space characters will have been subsumed into preprocessing tokens. White-space characters only have significance now when preprocessing tokens are glued together, or as a possible constraint violation (i.e., vertical-tab or form-feed within a preprocessing directive). 1952 white space between macro argument tokens 1864 white-space within preprocessing directive

Other Languages

Some languages treat the amount of white space at the start of a line as being significant (make requires a horizontal tab at the beginning of a line in some contexts[1329]). In Fortran (prior to Fortran 90) statements were preceded by six space characters (five if a line continuation was being used, or a comment). In Occam statement indentation is used to indicate the nesting of blocks.

Common Implementations

Most implementations replace multiple white-space characters by one space character. The existence, or not, of white-space separation can be indicated by a flag associated with each preprocessing token, preceded by space.

Integrated development environments vary in their handling of white-space. Some only allow multiple white-space characters, between tokens, at the start of a line, while others allow them in any context. White-space characters introduce complexity for tools' vendors[1434] that is not visible to the developer.

Coding Guidelines

Sequences of more than one white-space character often occur at the start of a line. They also occur between tokens forming a declaration when developers are trying to achieve a particular visual layout. However, white-space can only make a difference to the behavior of a program, outside of the contents of a character constant or string literal, when they appear in conjunction with the stringize operator. 1950 # operator

Example

```
#define mkstr(a) #a

char *p = mkstr(2  []; /* p may point at the string "2  [", or "2 [" */
char *q = mkstr(2[];   /* q points at the string "2[" */
```

translation phase 4

4. Preprocessing directives are executed, macro invocations are expanded, and **_Pragma** unary operator expressions are executed. 129

Commentary

This phase is commonly referred to as *preprocessing*. The various special cases in previous translation phases do not occur often, so they tend to be overlooked.

Although the standard uses the phrase *executed*, the evaluation of preprocessor directives is not dynamic in the sense that any form of iteration, or recursion, takes place. There is a special rule to prevent recursion from occurring. The details of macro expansion and the **_Pragma** unary operator are discussed elsewhere.

macro being replaced found during rescan 1970
macro replacement _Pragma operator 2030

C90

Support for the **_Pragma** unary operator is new in C99.

C++

Support for the **_Pragma** unary operator is new in C99 and is not available in C++.

Other Languages

PL/1 contained a sophisticated preprocessor that supported a subset of the expressions and statements of the full language. For the PL/1 preprocessor, *executed* really did mean executed.

Common Implementations

The output of this phase is sometimes written to a temporary source file to be read in by the program that implements the next phrase of translation.

If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. 130

Commentary

The C Standard allows UCNs to be interpreted and converted into internal character form either in translation phase 1 or translation phase 5 (the C committee could not reach consensus on specifying only one of these). If an implementation chooses to convert UCNs in translation phase 1, it makes no sense to require them to perform another conversion in translation phase 4. This behavior is different from that for other forms of preprocessing tokens. For instance, the behavior of concatenating two integer constants is well defined, as is concatenating the two preprocessing tokens whose character sequences are `0x` and `123` to create a hexadecimal constant.

universal character name syntax 815

The intent is that universal character names be used to create a readable representation of the source in the native language of the developer. Once this phase of translation has been reached, the sequence of characters in the source code needed to form that representation are not intended to be manipulated in smaller units than the universal character name (which may have already been converted to some other internal form).

C90

Support for universal character names is new in C99.

C++

translation phase 1 116

In C++ universal character names are only processed during translation phase 1. Character sequences created during subsequent phases of translation, which might be interpreted as a universal character name, are not interpreted as such by a translator.

Coding Guidelines

Support for universal character names is new and little experience has been gained in the kinds of mistakes developers make in its usage. It is still too early to know whether a guideline recommending "A character sequence that matches the syntax of a universal character name shall not be produced by token concatenation." is worthwhile.

Example

```
#define glue(a, b) a ## b

int glue(\u1, 234);
```

131 A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

Commentary

The **#include** preprocessing directives are processed in-depth first order. Once an **#include** has been fully processed, translation continues in the file that **#include**d it. There is a limit on how deeply **#include**s can be nested. Just prior to starting to process the **#include**, the macros __FILE__ and __LINE__ are set (and they are reset when processing resumes in the file containing the **#include**).

295 limit #include nesting
2007 __FILE__ macro
2008 __LINE__ macro

The effect of this processing is that phase 5 sees a continuous sequence of preprocessing tokens. These preprocessing tokens do not need to maintain any information about the source file that they originated from.

132 All preprocessing directives are then deleted.

preprocessing directives deleted

Commentary

This requirement was not explicitly specified in C90. In practice it is what all known implementations did. Also, macro definitions have no significance after translation phase 4.

1975 macro definition no significance after

Preprocessing directives have their own syntax, which does not connect to the syntax of the C language proper. The preprocessing directives are used to control the creation of preprocessing tokens. These are handed on to subsequent phases; they don't get past phase 4.

1854 preprocessor directives syntax

C++

This explicit requirement was added in C99 and is not stated in the C++ Standard.

Common Implementations

Many implementations write the output of the preprocessor to a file to be read back in by the next phase. Ensuring that declarations and statements retain the same line number as they had in the source file allows more specific diagnostic messages to be produced. To achieve this effect, some implementations convert preprocessing directives to blank lines, while others insert **#line** directives.

133 5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set;

translation phase 5

Commentary

The execution character set is used by the host on which the translated program will execute. Up until this phase characters have been represented using the physical-to-source character set mapping that occurred in translation phase 1.

218 execution character set represented by
116 translation phase 1

The translation and execution environments may, or may not, use the same values for character set members. This conversion relies on implementation-defined behavior. In the case of escape sequences, an implementation is required to use the value specified (provided this value is representable in the type **char**).

870 character constant mapped
873 escape sequence octal digits
875 escape sequence hexadecimal digits

For instance, the equality `'\07'==7` is true, independent of whether any member of the basic execution character set maps to the value seven. This issue is discussed in DR #040q8.

Common Implementations

Probably the most commonly used conversion uses the values specified in the Ascii character set. Some mainframe hosted implementations continue to use EBCDIC.

Coding Guidelines

Differences between the values of character set members in the translation and execution environments become visible if a relationship exists between two expressions, one appearing in a **`#if`** preprocessing directive and the other as a controlling expression. This issue is discussed elsewhere.

footnote 1874 141

Example

In the following:

```
1  void f(void)
2  {
3  char ch = 'a';
4  char dollar = '$';
5  }
```

the integer character constant `'a'` is converted to its execution time value. It may have an Ascii value in the source character set and an EBCDIC value in the execution character set. The $ character will have been mapped to the source character set in translation phase 1. In the following example it can be mapped back to the $ character if the implementation so chooses.

```
1  #if ('a' == 97) && ('Z' == 90)
2  #define PREPROCESSOR_USES_ASCII 1
3  #else
4  #define PREPROCESSOR_USES_ASCII 0
5  #endif
6
7  _Bool ascii_execution_character_set(void)
8  {
9  return ('a' == 97) && ('Z' == 90);
10 }
```

---

corresponding member if no

if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[7)] 134

Commentary

All source character set members and escape sequences, which have no corresponding execution character set member, may be converted to the same member, or they may be converted to different members. The behavior is required to be documented.

The null character is special in that it is used to terminate string literals. It is possible for a string literal to contain a null character through the use of an escape sequence, but such an occurrence has to be explicitly created by the developer. It is never added by the implementation.

C90

The C90 Standard did not contain this statement. It was added in C99 to handle the fact that the UCN notation supports the specification of numeric values that may not represent any specified (by ISO 10646) character.

ISO 10646 28

C++

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

116 translation phase 1

C++ handles implementation-defined character members during translation phase 1.

Common Implementations

Most implementations simply convert escape sequences to their numerical value. There is no check that the value maps to a character in the execution character set. Characters in the source character set, which are not in the execution character set, are often mapped to the value used to represent that value in the translation environment.

Coding Guidelines

Why would source code contain an instance of a source character or escape sequence that did not have a corresponding member in the execution character set? The usage could be because of a fault in the program (and therefore outside the scope of these coding guidelines), or existing source is being ported to a new environment that does not support it.

Use of members of an extended character set is by its very nature dependent on particular environments. Porting source that contains such character usage does not fall within the scope of these guidelines.

216 extended character set

Example

```
1  char g_c = '$';
2  char my_address = "derek@99.C.ISO.Earth";
3
4  char e_c = '\007';
5
6  #if '@' == 64
7  char *char_set = "ASCII";
8  #else
9  char *char_set = "EBCDIC ;-)";
10 #endif
```

---

135 6. Adjacent string literal tokens are concatenated.

translation phase 6

Commentary

This concatenation only applies to string literals. It is not a general operation on objects having an array of **char**. String literal preprocessing tokens do not have a terminating null character. That is added in the next translation phase.

C90

6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

It was a constraint violation to concatenate the two types of string literals together in C90. Character and wide string literals are treated on the same footing in C99.

The introduction of the macros for I/O format specifiers in C99 created the potential need to support the concatenation of character string literals with wide string literals. These macros are required to expand to character string literals. A program that wanted to use them in a format specifier, containing wide character string literals, would be unable to do so without this change of specification.

Other Languages

Most languages that support string concatenation require that the appropriate operator be used to specify an operation to be performed. For instance, Ada uses the character **&**.

Coding Guidelines

There is the possibility, in an initializer or argument list, that a missing comma causes two unrelated string literals to be concatenated. However, this is a fault and considered to be outside the scope of these coding guidelines.

guidelines 0 not faults

Example

```
#include <stdio.h>

/*
 * Assign the same value, L"ab", to three different objects.
 */
wchar_t *w_p1 = L"a" L"b",
        *w_p2 = L"a"  "b",
        *w_p3 = "a" L"b";

void f(void)
{
printf("\\" "066"); /* Output \066, escape sequences were processed earlier. */
}
```

Usage

In the visible form of the `.c` files 4.9% (`.h` 15.6%) of string literals are concatenated.

---

translation phase 7

7. White-space characters separating tokens are no longer significant. 136

Commentary

White-space is not part of the syntax of the C language. It is only significant in separating characters in the lexical grammar and in some contexts in the preprocessor. This statement could have equally occurred in translation phase 5.

replacement list 1918 identical if

Other Languages

Fortran 777 spaces not significant

In Fortran white space is never significant.

Example

```
#define mkstr(x) #x

char *a_space_plus_b = mkstr(a +b);
char *a_plus_b = mkstr(a+b);
```

---

preprocessing token converted to token

Each preprocessing token is converted into a token. 137

Commentary

These are the tokens seen by the C language parser (the same conversion also occurs within **#if** preprocessing directives). It is possible for a preprocessing token to not be convertible to a token. For instance, in:

#if 1878 identifier replaced by 0

```
float f = 1.2.3.4.5;
```

`1.2.3.4.5` is a valid preprocessing token; it is a pp-number. However, it is not a valid token. 927 pp-number syntax

Preprocessing tokens that are skipped as part of conditional compilation need never be converted to tokens (because they never make it out of translation phase 4).

```
#if 0
float f = 1.2.3.4.5; /* Never converted. */
#endif
```

A preprocessing token that cannot be converted to a token is likely to cause a diagnostic to be issued. At the very least, there will be a syntax violation. It is a quality-of-implementation issue as to whether the translator issues a diagnostic for the failed conversion.

Other Languages

Few other language contain a preprocessor. Any problems associated with creating a token are directly caused by the sequence of input characters.

---

138 The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. syntactically analyzed

Commentary

This is what the bulk of the standard's language clause is concerned with.

Common Implementations

This is the phase where the executable code usually gets generated.

Luo, Chen, and Yu[877] found that an equation of the form L^x, where L is the number of bytes in the source file (i.e., header file contents are not included) and x some constant, provided a good estimate for the elapsed time needed to translate a source file (an x value of approximately 0.66 was found for GCC 3.4.2) and to link it (an x value of approximately 0.1 was found for GCC). 108 source files

Coding Guidelines

Coding guidelines are not just about how the semantics phase of translation processes its input. Previous phases of translation are also important, particularly preprocessing. The visible source, the input to translation phase 1, is probably the most important topic for coding guidelines.

---

139 8. All external object and function references are resolved. translation phase 8

Commentary

This resolution is normally carried out by a program commonly known as a linker. Although the standard says nothing about what such resolution means its common usage (in linker terminology) is that references to declarations are made to refer to their corresponding definitions.

C++

The C translation phase 8 is numbered as translation phase 9 in C++ (in C++, translation phase 8 specifies the instantiation of templates).

Common Implementations

The code generated for a single translation unit invariably contains unresolved references to external objects and functions. The tool used to resolve these references, a linker, may be provided by the implementation vendor or it may have been supplied as part of the host environment.

Resolving references that involve addresses larger than the length of a machine code instruction can lead to inefficiencies and wasted space. For instance, generating a 32-bit address using instructions that have a maximum length of 16 bits requires at least three instructions (some bits are needed to specify what the instruction does; for instance, load constant). In practice most addresses do not require a full 32-bit value, but this information is not available until link-time, while translators have been forced to make worst-case assumptions. Many processors solve this problem by having variable-length instructions.

The simplifications derived from the principles behind RISC mean that their instructions have a fixed length. It is possible to handle 32 bit addresses using a 32-bit instruction width by appropriate design conventions (e.g., the call instruction encoded in 2 bits, leaving 30 bits for addresses and requiring that functions start on a 4-byte address boundary). The move to a 64-bit address space reintroduces problems seen in the 16/32-bit era two decades earlier. However, processor performance and storage capabilities have moved on and link-time optimizations are now practical. In particular the linker knows the values of addresses and can generate the minimum number of instructions needed.[1276]

The quest for higher-quality machine code has led research groups to look at link-time optimizations.[1437] Having all of the components of a program available opens up opportunities for optimizations that are not available when translating a single translation unit. A study by Muth, Debray, Watterson, and De Bosschere[985] found that on average 18% of a programs instructions could have their operands and result determined at link time. This does not imply that 18% of a program's instructions could be removed; knowing information provides opportunities for optimization, like replacing an indirect call with a direct one.

Levine[848] discusses the principles behind linking object files to create a program image and loading that program image, into storage, prior to executing it.

Coding Guidelines

The problem of ensuring that the same identifier, declared in different translation units, always resolves to a definition having the same type is discussed elsewhere.

identifier declared in one file 422.1

Library components are linked to satisfy external references to functions and objects not defined in the current translation. 140

Commentary

The term *library components* is a broad one and can include previously translated translation units. The implementation is responsible for linking in any of its provided C Standard library functions, if needed.

translation units preserved 111

The wording of this requirement can be read to imply that all external references must be satisfied. This would require definitions to exist for objects and functions, even if they were never referenced during program execution (but were referenced in the translated source code). This is the behavior enforced by most linkers.

Since there is no behavior defined for the case where a reference cannot be resolved, it is implicitly undefined behavior.

undefined behavior indicated by 85

Other Languages

This linking phase is common to many implementations of most programming languages that support some form of separate compilation. In a few cases the linking process can occur on an as-needed basis during program execution, as for instance in Java.

translation unit syntax 1810

Common Implementations

linkers

Most linkers assume that all of the definitions in the developer-supplied list of object files, given to the linker, are needed, but that definitions from the implementation system libraries only need be included if they are referenced from the developer-written code.

Optimizing linkers build a function call tree, starting from the function `main`. This enables them to exclude functions, supplied by the developer, from the program image that are never referenced during execution.

The two commonly used linking mechanisms are static and dynamic linking. Static linking creates a program image that contains all of the function and object definitions needed to execute the program. A statically linked program image has no need for any other library functions at execution time. This method of linking is suitable when distributing programs to host environments where very little can be assumed about the availability of support libraries. If many program images are stored on a host, this method has the disadvantage of keeping a copy of library functions that are common to a lot of programs. Also if any libraries are updated, all programs will need to be relinked.

Dynamic linking does not copy library functions into the program image. Instead a call to a dynamic linking system function is inserted. When a dynamically linked function is first called, the call is routed

via the dynamic link loader, which resolves the reference to the desired function and patches the executing program so that subsequent calls jump directly to that routine. Provided there is support from the host OS, all running programs can access a single copy of the library functions in memory. On hosts with many programs running concurrently, this can have a large impact on memory performance (reduced swapping). Any library updates will be picked up automatically. The program image lets the dynamic loader decide which version of a library to call.

An implementation developed by Neamtiu, Hicks, Stoyle and Oriol[999] supports what they call *dynamic software updating*, whereby an executing program can be patched at a fine level of granularity, e.g., a modified version of a function (the parameter and return types have to be compatible) can replace the one currently in the executing program image.

Coding Guidelines

The standard does not define the order in which separately translated translation units and implementation supplied libraries are scanned to resolve external references. Many implementations scan files in the order in which they are given to the linker.

Most implementations define additional library functions. The names of such functions rarely follow the conventions, defined in the C Standard, for implementation-defined names. There is the possibility that one of these names will match that of an externally visible, developer-written function definition. Placing system libraries last on the list to be processed helps to ensure that any developer-provided definitions, whose names match those in system libraries, are linked in preference to those in the system library. Of course there may then be the problem of the system library referencing the incorrect definition.

Linking is often an invisible part of building the program image (diagnostics are sometimes issued for multiply defined identifiers). Checking which references have been resolved to which definitions is often very difficult. In the case of an incorrect function definition being used, a set of tests that exercise all called functions is likely to highlight any incorrect usage. In the case of incorrect objects, things might appear to work as expected. A guideline recommendation dealing with the incorrect definition being used to build the program image sounds worthwhile. However, this usage is likely to be unintended (a fault) and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

0 guidelines not faults

---

141 All such translator output is collected into a program image which contains information needed for execution in its execution environment.

program image

Commentary

This wording implies a single location, perhaps a file (containing all of the information needed) or a directory (potentially containing more than one file with all of the necessary information). Several requirements can influence how a program image is built, including the following:

- *Developer convenience.* Like all computer users, developers want things to be done as quickly as possible. During the coding and debugging of a program, its image is likely to be built many times per hour. Having a translator that builds this image quickly is usually seen as a high priority by developers.
- *Support for symbolic debugging.* Here information on the names of functions, source line number to machine code offset correspondence, the object identifier corresponding to storage locations, and other kinds of source-related mappings needs to be available in the program image. Requiring that it be possible to map machine code back to source code can have a significant impact on the optimizations that can be performed. Research on how to optimize and include symbolic debugging information has been going on for nearly 20 years. Modern approaches[1485] are starting to provide quality optimizations while still being able to map code and data locations.
- *Speed of program execution.* Users of applications want them to execute as quickly as possible. The organization of a program image can have a significant impact on execution performance.

- *Hiding information.* Information on the internal workings of a program may be of interest to several people. The owner of the program image may want to make it difficult to obtain this information from the program image.[1440]
- *Distrust of executable programs.* Executing an unknown program carries several risks: It may contain a virus, it may contain a trojan that attempts to obtain confidential information, it may consume large amounts of system resources or a variety of other undesirable actions. The author of a program may want to provide some guarantees about a program, or some mechanism for checking its integrity. There has been some recent research on translated programs including function specification and invariant information about themselves, so-called proof carrying programs.[1001] Necula's proposed method includes a safety policy and operating in the presence of untrusted agents. On the whole the commonly used approach is for the host environment to ring fence an executing program as best it can, although researchers have started to look at checking program images[1488] before loading them, particularly into a trusted environment.

Other Languages

Most language specifications are silent on the issue of building program images. The Java specification explicitly specifies a possible mechanism supporting a distributed, across separate files, program image.

Common Implementations

Some program images contain all of the necessary machine code and data; they don't reference other files containing additional definitions— static linking. While other program images copy the object code derived from developer-written code in a single file, which also contains references to system-supplied definitions that need to be supplied during execution (held in other files)— dynamic linking.

The file used to hold a program image usually has some property to indicate its status as an executable program. Under MS-DOS the name the file extension is used (`.exe` or `.com`). Under Unix the list of possible file attributes, held by the file system, includes a flag to indicate an executable program.

The information in a program image may be laid out so that it can be copied directly into memory, as is. For instance, a MS-DOS `.com` file is completely copied into memory, starting at address 0x100. A Unix ELF (Executable and Linking Format) file contains segments that are copied to memory, as is (additional housekeeping information is also held in the file). Arranging for a program image to contain an exact representation of what needs to be held in memory has the advantage of simplifying swapping (assuming that the host performs this relatively high-level memory-management function) of unused code. It can simply be read back in from the program image; it does not have to be copied from memory to a swap partition and read back in.

On some hosts the layout of the program image, on a storage device, is different from its execution-time layout in storage. In this case the information in the program image is used to construct the execution-time layout when the program is first invoked. Such image formats include COFF (Common Object File Format) under Unix, MS-DOS `.exe` files and the IBM 360 object format (based on fixed-length records of 80 characters).

The ANDF (Architecture Neutral Distribution Format) project, http://www.tendra.org, aimed to produce a program image that could be installed on a variety of different hosts. The image contained a processed form of the original source code. An installer, running on a particular host, took this intermediate program image and a platform-specific program image (containing the appropriate machine code) from it, thus installing the program on that host. Installers were produced for a number of different host processors. The technology has not thrived commercially and much of the source code has now been made publicly available.

The performance of an application during certain parts of its execution is sometimes more important than during other parts. Program startup is often one such instance. Users of applications often want them to start as quickly as possible. How the program image is organized can affect the time taken for a program to start executing. One study by Lee, Crowley, Baer, Anderson, and Bershad[819] found that program startup

only required a subset of the information held in the program image. They also found that in some cases applications had scattered this information over the entire program image, requiring significantly more information to be read than was required for startup. They showed that by reordering how the program image was structured it was possible to reduce application startup latency by more than 58% (see Table 141.1).

Table 141.1: *Total* is the number of code pages in the application; *Touched* the number of code pages touched during startup; *Utilization* the average fraction of functions used during startup in each code page. Adapted from Lee.[819]

Application	Total	Touched (%)	% Utilization
acrobat	404	246 (60)	28
netscape	388	388 (100)	26
photoshop	594	479 (80)	28
powerpoint	766	164 (21)	32
word	743	300 (40)	47

Program images contain sequences of machine code instructions that are executed by the host processor. In some application domains the instructions making up a program can account for a large percentage of memory usage. Several techniques are available to reduce this overhead: Compression techniques can be applied to the sequence of instructions, or function abstraction can be applied.

instruction compression

A variety of instruction compression schemes have been proposed. The optimal choice can depend on the processor instruction format (RISC processors with fixed-length instructions, or DSP processors using very wide instruction words) and whether decompression is supported in hardware or software.[828] Compression ratios in excess of 50% are commonly seen. Performance penalties on decompression range from a few percent to 1,000% (for software decompression). At the time of this writing IBM's PowerPC 405 is the only commercially available processor that makes use of hardware compression (the CodePack algorithm). A good discussion of the issues and experimental results can be found in[829]

On early Acorn RISC Unix machines, the bottleneck in loading a program image was the disk drive. Analysis showed that it was quicker to load a smaller image and decompress it, on the fly, than to fetch the extra disk blocks with no decompression (a simple compression program was included with each system). The encoding of the ARM machine code created opportunities for compression. For instance, the first 4 bits of each opcode are a conditional. Most instructions are always executed and these four bits contained the value 0xE; a value of 0xF meant *never* and rarely occurred in the image.

Another approach to reducing the memory footprint of a program is to use compression on a function-by-function basis. Kirovski[739] reported an average memory reduction of 40% with a runtime performance overhead of 10% using a scheme that held uncompressed functions in a cache; a call to a compressed function causes it to be uncompressed and placed in the cache, possibly removing the uncompressed code for a function already in the cache.

The ordering of functions within the program image can also affect performance by affecting what is loaded into any instruction cache. In one study[1198] it was found that in an Oracle database running an online transaction-processing application (with an instruction footprint of about 1 M byte) 25% to 30% of the execution time was caused by instruction stalls. By reordering functions in memory (dynamically, during program execution, based on runtime behavior), it was possible to improve performance by 6% to 10%[1198]. 1710 basic block

function abstraction

Function abstraction (sometimes called *procedure abstraction*) is the process of creating functions from sequences of machine instructions. A call to this translator-created function replaces the sequence of instructions that it duplicates. On its own this process would slow program execution and make the program image larger. However, if the same sequence of instructions occurs at other places in the program, they can also be replaced by a call. Provided the sequence being replaced is larger than the substituted call instruction, the program image will shrink in size. Zastre[1501] provides a discussion of the issues.

For a collection of embedded applications, Cooper and McIntosh[276] were able to reduce the number of instructions in the program image by 4.88%, with a corresponding increase of executed instructions of 6.47%.

Using profile information, they were able to reduce the dynamic overhead to 0.86%, with a corresponding decrease in the static instruction count of 3.89%.

Many optimization techniques will result in the same sequence of source statements being translated into different sequences of machine instructions. Context can be an important issue in optimization. Performing function abstraction prior to some optimizations can increase the number of duplicate sequences of instructions.[1189]

Coding Guidelines

Use of static linking may be thought to ensure that program images continue to produce the same behavior when a host's libraries are updated. Experience shows that changes to these libraries can cause changes of behavior, particularly when low-level device drivers are involved.

Use of dynamic linking ensures that programs can take advantage of any bug fixed or performance improvements associated with updated libraries. Use of dynamically linked libraries also helps to ensure that all programs use the same libraries (reducing the likelihood of different programs failing to cooperate because of execution-time incompatibilities, for instance through use of a shared memory interface). If a required library is not available in the host environment, either it will not be possible to execute the program image or it will fail during execution.

The issue of static versus dynamic linking is a configuration-management issue and is outside the scope of these coding guidelines.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9). 142

---

footnote 6

6) As described in 6.4, the process of dividing a source file's characters into preprocessing tokens is context-dependent. 143

Commentary

In only a few cases does a lexer, for C language, need information on the context in which it is encountering characters. The issue of whether an identifier is defined as a typedef or not is not applicable during the creation of preprocessing tokens; both have the same lexical syntax. However, the syntax used by most translators makes a distinction between identifiers and typedef names, so a symbol table lookup is needed when the preprocessing token is converted to a token (or at some other time, prior to translation phase 7).

Other Languages

In most languages no knowledge of context is needed to create tokens from a character sequence. It has become accepted practice in language design that it be possible to implement a lexer using a finite state automata. There are several tools that automatically produce lexers from a specification; for example, `flex` from the Open Software Foundation.

In Fortran white space is not significant and in some cases it is necessary to scan ahead of the current input character to decide the lexical form of a given sequence of characters.

Fortran 777 spaces not significant

For example, see the handling of < within a **`#include`** preprocessing directive. 144

Commentary

This issue is discussed elsewhere.

header name 924 recognized within #include

---

footnote 7

7) An implementation need not convert all non-corresponding source characters to the same execution character. 145

Commentary

An implementation may choose to convert all source characters without a corresponding member in the execution character set to a single execution character whose interpretation implies, for instance, that there was no corresponding member. Your author knows of no implementation that takes this approach.

correspond-ing member 134 if no

C++

The C++ Standard specifies that the conversion is implementation-defined (2.1p1, 2.13.2p5) and does not explicitly specify this special case.

5.1.1.3 Diagnostics

146 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined.

diagnostic shall produce

Commentary

This is a requirement on the implementation. The first violation may put the translator into a state where there are further, cascading violations. The extent to which a translator can recover from a violation and continue to process the source in a meaningful way is a quality-of-implementation issue.

The standard says nothing about what constitutes a diagnostic message (usually simply called a *diagnostic*). Although each implementation is required to document how they identify such messages, playing a different tune to represent each constraint or syntax violation is one possibility. Such an implementation decision might be considered to have a low quality-of-implementation.

65 diagnostic message

The Rationale uses the term *erroneous program* in the context of a program containing a syntax error or constraint violation. Developers discussing C often use the term *error* and *erroneous*, but the standard does not define these terms.

C++

> *— If a program contains a violation of any diagnosable rule, a conforming implementation shall issue at least one diagnostic message, except that*
>
> *— If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.*

1.4p2

A program that contains "a violation of a rule for which no diagnostic is required", for instance on line 1, followed by "a violation of any diagnosable rule", for instance on line 2; a C++ translator is not required to issue a diagnostic message.

Other Languages

Java contains no requirement that any violations of its compile-time requirements be diagnosed.

Common Implementations

Traditionally C compilers have operated in a single pass over the source (or at least one complete pass for preprocessing and another complete pass for syntax and semantics, combined), with fairly localized error recovery.

10 implementation single pass

Constraint violations during preprocessing can be difficult to localize because of the unstructured nature of what needs to be done. If there is a separate program for preprocessing, it will usually be necessary to remove all constraint violations detected during preprocessing. Once accepted by the preprocessor the resulting token stream can be syntactically and semantically analyzed. Constraint violations that occur because of semantic requirements tend not to result in further, cascading, violations.

Syntax violations usually causing a message of the form "Unexpected token", or "One of the following . . . was expected" to be output. Recovering from syntax violations without causing some additional, cascading violations can be difficult.

161 translated invalid program

147 Diagnostic messages need not be produced in other circumstances.[8)]

diagnostic message produced other circumstances

Commentary

The production of diagnostics in other circumstances is a quality-of-implementation issue. Implementations are free to produce any number of diagnostics for any reason, but they are not required to do so.

Common Implementations

Over time the number of diagnostics produced by implementations has tended to increase. Minimalist issuance of diagnostics is not considered good translator practice.

Coding Guidelines

Some translators contain options to generate diagnostics for constructs that are they consider suspicious, or a possible mistake on the developer's part. Making use of such options is to be recommended as a way of helping to find potential problems in source code.

A guideline recommendation of the form "The translator shall be run in a configuration that maximizes the likelihood of it generating useful diagnostic messages." is outside the scope of these coding guidelines. A guideline recommendation of the form "The source code shall not cause the translator to generate any diagnostics, or there shall be rationale (in a source code comment) for why the code has not been modified to stop them occurring." is also outside the scope of these coding guidelines.

EXAMPLE constraint violation and undefined behavior

EXAMPLE An implementation shall issue a diagnostic for the translation unit: 148

```
char i;
int i;
```

because in those cases where wording in this International Standard describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error shall be diagnosed.

Commentary

An implementation is not allowed to define undefined behavior to be: No diagnostic is issued.

5.1.2 Execution environments

Two execution environments are defined: *freestanding* and *hosted*. 149

Commentary

Freestanding is often referred as an *embedded system*, outside of the C Standard's world.

Other Languages

Most languages are defined (often implicitly) to operate in a hosted environment. The Java environment has a single, fully specified environment, which must always be provided.

Common Implementations

Vendors tend to sell their products into one of the two environments. The `gcc` implementation has been targeted to both environments.

program startup

In both cases, *program startup* occurs when a designated C function is called by the execution environment. 150

Commentary

Executing the developer-written functions is only part of the process of executing a program. There are also various initialization and termination activities (discussed later).

The C Standard does not deal with situations where there may be more than one program executing at the same time. Its model is that of a single program executing a single thread of execution.

Common Implementations

Implementations use one of two techniques to call the designated function (called `main` here):

```
1   result_code=main(argc, argv);
```

or,

```
1   exit(main(argc, argv));
```

The method used to make the call does not affect startup, but it can affect what happens when control is handed back to the startup function by the program.

---

151 All objects with static storage duration shall be *initialized* (set to their initial values) before program startup.

static storage duration initialized before startup

Commentary

This is a requirement on the implementation.

This initialization applies to all objects having file scope and objects in block scope that have internal linkage. The initial values may have been provided explicitly by the developer or implicitly by the implementation. Once set, these objects are never reinitialized again during the current program invocation, even if `main` is called recursively (permitted in C90, but not in C99 or C++).

455 static storage duration
1677 object initialized but not explicitly

C++

In C++ the storage occupied by any object of static storage duration is first zero-initialized at program startup (3.6.2p1, 8.5), before any other initialization takes place. The storage is then initialized by any nonzero values. C++ permits static storage duration objects to be initialized using nonconstant values (not supported in C). The order of initialization is the textual order of the definitions in the source code, within a single translation unit. However, there is no defined order across translation units. Because C requires the values used to initialize objects of static storage duration to be constant, there are no initializer ordering dependencies.

Other Languages

Some languages (e.g., the original Pascal Standard) do not support any mechanism for providing initial values to objects having static storage duration. Java allows initialization to be delayed until the first active use of the class or interface type. The order of initialization is the textual order of the definitions in the source code.

Common Implementations

Using the as-if rule to delay initialization until the point of first reference often incurs a high execution-time performance penalty. So most implementations perform the initialization, as specified, prior to startup.

In freestanding environments where there are warm resets, realtime constraints on startup and other constraints sometimes mean that this initialization is omitted on startup. A program can rely on objects being explicitly assigned values through executed statements (appearing in the source code).

The IAR PICmicro compiler[612] supports the `__no_init` specifier which causes the translator to not generate machine code to initialize the so-designated objects on program startup.

Coding Guidelines

It is guaranteed that all objects having file scope will have been initialized to some value prior to program startup. Some coding guideline documents require that all initializations be explicit in the source code and implicit initialization not be relied on. However, it is unclear what the proposers of such a guideline recommendation are trying to achieve. Experience suggests that these authors are applying rationales whose cost/benefit is only worthwhile for objects defined in block scope.

use initializers discussion
1641 initialization syntax

---

152 The manner and timing of such initialization are otherwise unspecified.

initialization timing

Commentary

For objects defined at file scope the standard only supports initialization expressions whose value is known at translation-time. This means that there are no dependencies on the order of these initializations. It is not possible to write a conforming C program that can deduce the manner and timing of initialization.

1644 initializer static storage duration object

Floating-point constants, the representation of which can vary between hosts, may be stored in a canonical form in the program image, and be converted by startup code to the representation used by the host.

Common Implementations

The manner in which initialization occurs can take several forms. The program image may simply contain a list of values that need to be copied into memory; many linkers merge the initializations of sequences of contiguous memory locations into a single block, this exact memory image being copied on startup. For large blocks of memory being initialized to a single value, translators may generate code that loops over those locations, explicitly setting them. Some host environments set the memory they allocate to programs to zero, removing the need for much initialization on behalf of the developer, which for large numbers of locations is zero.

linkers 140

Addresses of objects that have been assigned to pointers may have to be fixed up during this initialization (information will have been stored in the program image by the linker).

In a freestanding environment values may be held in ROM, ready for use on program startup.

Coding Guidelines

Use of the C++ functionality of having nonconstant expressions for the initializers of static storage duration objects may be intended or accidental. Using this construct intentionally is making use of an extension. It may also be necessary to be concerned with order of initialization issues (the order is defined by the C++ Standard).

The use may be accidental because some C++ compilers do not diagnose such usage when running in their C mode. Checking the behavior of the translator using a test case is straight-forward. The action taken, if any, in those cases where the use is not diagnosed will depend on the cost of independent checking of the source (via some other tool, even a C translator) and the benefit obtained. This is a management, not a coding guidelines, issue.

Example

```
1  extern int glob_1;
2
3  static int sglob = glob_1; /* not conforming */
4                             // conforming
```

Program termination returns control to the execution environment. 153

Commentary

freestanding 157
environment
program termination

There are many implications hidden beneath this simple statement. Just because control has been returned to the execution environment does not mean that all visible side effects of executing that program are complete. For instance, it may take some time for the output written to files to appear in those files.

Program termination also causes all open files to be closed.

C++

3.6.1p1 *[Note: in a freestanding environment, start-up and termination is implementation defined;*

3.6.1p5 *A* **`return`** *statement in* `main` *has the effect of leaving the main function (destroying . . . duration) and calling* `exit` *with the return value as the argument.*

18.3p8

The function `exit()` *has additional behavior in this International Standard:*

. . . Finally, control is returned to the host environment.

Common Implementations

In a hosted environment program, termination may, or may not, have a large effect on the execution environment. If the host does not support concurrent execution of programs, termination of one program allows another to be executed.

In a hosted environment the call to the designated function, when startup occurs, usually makes use of a stack that has already been created by the host environment as part of its process of getting a program image ready to execute. Program termination is the point at which this designated function returns (or a call to one of the library functions `exit`, `abort`, or `_Exit` causes the implementation to unwind this stack internally). Control then returns to the code that performed the original invocation.

---

154 **Forward references:** storage durations of objects (6.2.4), initialization (6.7.8).

5.1.2.1 Freestanding environment

Commentary

As little as possible is said about freestanding environments, since little is served by constraining them. Rationale

---

155 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. freestanding environment startup

Commentary

The function called at program startup need not be spelled `main`.

The parameters to the called function are defined by the implementation and can be very different from those defined here for `main` (in a hosted environment). If there is no operating system, there are not likely to be any command line parameters, but information may be passed into the function called.

Without the benefit of an operating system, the issue of how to provide the ability to startup different programs becomes important. Having available a set of functions, which can be selected amongst just prior to program startup, provides one solution.

Common Implementations

In many cases there is no named program at all. Switching on, or resetting, the freestanding host causes the processor instruction pointer to be set to some predefined location in storage, whichever instructions are located at that and subsequent locations being executed. Traditionally there is a small bootstrap loader at this location, which copies a larger program from a storage device into memory and jumps to the start of that program (it might be a simple operating system). In other cases that storage device is ROM and the program will already be in memory at the predefined location. Translated program instructions are executed directly from the storage device. Once switched on, the host may have buttons that cause the processor to be reset to a different location, causing different functions to be invoked on startup.

---

156 Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined. environment freestanding implementation

Commentary

conforming 94 freestanding implementation

The Committee wanted it to be practical to create a conforming implementation for a freestanding environment. In such an environment the basic facilities needed to provide all of the functionality required by the full library are often not available. The requirement to supply a full library needed to be relaxed. The full language can always be supported, although this may mean supplying a set of internal functions for handling arithmetic operations on **long long** and floating-point types. If a particular program does not make use of **long long** or floating-point types, a linker will usually take the opportunity to reduce the size of the final executable by not linking in these internal library functions.

Common Implementations

Freestanding environments vary enormously between those that offer a few hundred bytes of storage and no library, to those having megabytes of memory and an operating system offering full POSIX realtime functionality.

Coding Guidelines

Care needs to be taken, in a freestanding environment, when using library facilities beyond the minimal set required by Clause 4. There is no requirement for an implementation that supplies additional functions to completely follow the specification given in the standard. It is not uncommon to see only partial functionality implemented.

---

freestanding environment program termination

The effect of program termination in a freestanding environment is implementation-defined. 157

Commentary

It may not even be possible to terminate a program in a freestanding environment. The program may be the only execution environment there is. Switching the power off may be the only way of terminating a program.

Common Implementations

A **return** statement executed from the function called on program startup, or a call to `exit` (if the freestanding implementation supports such a call) may return control to a host operating system or return to the random address held on the top of the stack.

Coding Guidelines

Specifying how program termination, in a freestanding environment, should be handled is outside the scope of these coding guidelines.

5.1.2.2 Hosted environment

---

hosted environment

A hosted environment need not be provided, but shall conform to the following specifications if present. 158

Commentary

This is a requirement on the implementation. The standard defines a specification, not an implementation. Issues such as typing the name of the program on a command line or clicking on an icon are possible implementations, which the standard says nothing about.

C++

1.4p7 *For a hosted implementation, this International Standard defines the set of available libraries.*

17.4.1.3p1 *For a hosted implementation, this International Standard describes the set of available headers.*

Common Implementations

Most hosted environments provide the full set of functionality specified here. The POSIX (ISO/IEC 9945) standard defines some of the functions in the C library. On the whole the specification of this functionality is a pure extension of the C specification.

Coding Guidelines

This specification is the minimal set of requirements that a program can assume will be provided by a conforming hosted implementation. The application domain may influence developer expectations about the minimal functionality available. This issue is discussed elsewhere.

0 coding guidelines applications

---

159 8) The intent is that an implementation should identify the nature of, and where possible localize, each violation.

footnote 8

Commentary

The Committee is indicating their intent here; it is not a requirement of the standard that this localization be implemented.

The US government has also had something to say on this issue: FIPS PUB 160 (Federal Information Processing Standards PUBlication), issued March 13, 1991, said:

> *The message provided will identify:*
>
> *— The statement or declaration that directly contains the nonconforming or obsolete syntax.*

FIPS PUB 160, 10pc

Paragraph c also included a fuller definition and requirements. A change notice, issued on August 24 1992, changed these requirements to:

> *Page 3, paragraph 10.c, Specifications: remove paragraph 10.c.*

FIPS PUB 160 no longer contains any requirements on the localization of violations in C source by implementations.

Common Implementations

Violations detected during the expansion of a macro invocation can be difficult to localize beyond the original name of the macro being expanded and perhaps the line causing the original invocations.

macro replacement

Many implementations handle C syntax by using a LALR(1) grammar to control an associated parser. Grammars having this property are guaranteed to be able to localize a syntax violation on the first unexpected token[9] (i.e., no possible sequence of tokens following it could produce a valid parse).

Violations arising out of semantic constraints are usually localized within a few tokens.

Translators tend to generate a diagnostic based on a localized cause for the constraint violations. Experience has shown that assuming a localized cause for a violation is a good strategy (it is often also the simplest to implement). For instance, in:

```
1  static char g;
2
3  void f(void)
4  {
5  char *v;
6
7  if (g)
8     {
9     int v;
10      /*
11       * Other code.
12       */
13    v = &g;
14    }
15 }
```

the assignment to v is likely to be diagnosed as an incompatible assignment, rather than a misplaced **}** token. Many violations do have a localized cause. Looking for a nonlocalized cause requires a lot more effort on the part of vendors' implementations and is only likely to improve the relevance of diagnostics in a few cases. Most implementations generate diagnostic messages that specify what the violation is, not what needs to be done to correct it. In:

```
1  struct {
2          int mem;
3         } s;
4  int glob;
5
6  void f(void)
7  {
8  int loc;
9
10 loc = s + mem;
11 }
```

the implementation diagnostic is likely to point out that the binary plus operator cannot be applied to an operand having a structure type. One of the skills learned by developers is interpreting what different translators' diagnostic messages mean.

Example

```
1  #define COMPARE(x, y) ((y) < (x))
2
3  struct {
4          int mem;
5         } s;
6  int glob;
7
8  void f(void)
9  {
10 int loc;
11
12 /*
13  * Where should an implementation point, on the following line,
14  * as an indication of where the violations occurred?
15  */
16 loc = COMPARE(glob, s);
17 }
```

Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. 160

Commentary

A valid program, as in a strictly conforming program.

C++

The C++ Standard does not explicitly give this permission. However, producing diagnostic messages that the C++ Standard does not require to be generated might be regarded as an extension, and these are explicitly permitted (1.4p8).

Other Languages

Most languages are silent on the topic of extensions. Languages that claim to have Java as a subset have been designed and compilers written for them.

Common Implementations

One syntax, or constraint violation, may result in diagnostic messages being generated for tokens close to the original violation. These may be caused by a translators' incorrect attempts to recover from the first violation, or they may have the same underlying cause as the first violation. In many cases these are localized and translation usually continues (on the basis that there may be other violations and developers would like to have a complete list of them).

Some translators support options whose purpose is to increase the number of messages generated. It is intended that these messages be of use to developers. For instance, the `gcc` option *-Wall* issues diagnostics for constructs which it considers to be potential faults. Some translators support an option that causes any usage of an extension, provided by the implementation, to be diagnosed.

100 implementation document

There is a market for tools that act like translators but do not produce a program image. Rather, they analyze the source code looking for likely coding and portability problems— so-called *static analysis tools*. The term *lint-like tools* is often heard, after the first such tool of their ilk, called `lint` (lint being the fluff-like material that clings to clothes and finds its way into cracks and crevices).

Implementation vendors receive a lot of customer requests for improvements in the performance of generated code and support for additional library functionality. The quality of diagnostic messages produced by translators is rarely given a high priority by purchasers. Experience shows that developers can adapt themselves to the kinds of diagnostics produced by a translator.

Example

Many translators now attempt to diagnose likely programmer errors. The classic example is:

```
1   if (x = y)
```

where:

```
1   if (x == y)
```

had been intended.

Experience suggests that developers will ignore such diagnostics, or disable them, if they are incorrect more than 40% of the time.

---

161 It may also successfully translate an invalid program.

translated invalid program

Commentary

The term *invalid program* is not defined by the C Standard. Common usage suggests that a program containing violations of syntax or constraints was intended.

Before the spread of desktop computers, it was common for programs to be translated via a batch process on a central computer. The edit/compile/run cycle could take an hour or more. Having the translator fail to translate a program, even if it contained errors, was often considered unacceptable. Useful work could often be done with an invalid (only a part of it was likely to be invalid) but executable program while waiting for the next cycle to complete. Within multiperson projects a change made by one person can have widespread consequences. Having a translator continue to operate, while a particular issue is resolved, often increases its perceived quality.

Common Implementations

There have been several parsers that perform very sophisticated error recovery.[337] In the case of the C language, inserting a semicolon token is a remarkably effective, and simple, recovery strategy from a syntax violation (another is deleting tokens up to and including the next semicolon).

Prior to the invention of desktop computers and even before the spread of minicomputers, when translation requests had to be submitted as a batch job, vendors found it worthwhile investing effort in sophisticated syntax error recovery.[282, 337] The turnaround of a batch-processing system being sufficiently long (in the

past, several hours was not considered unusual) that anything a translator could do to repair simple developer typing mistakes, in order to obtain a successful program build, was very welcome. Once the turnaround time on submitting source for translation became almost instantaneous, by providing developers with direct access to a computer, customer demand for sophisticated syntax error recovery disappeared.

Commonly recovered semantic errors include the use of undeclared objects (many implementations recover by declaring them with type **int**) or missing structure members (adding them to the structure type is a common recovery strategy).

Coding Guidelines

A guideline recommendation of the form "The translated output from an invalid program shall never be executed in a production environment." is outside the scope of the guidelines.

5.1.2.2.1 Program startup

hosted environment startup

The function called at program startup is named **main**. 162

Commentary

That is, the user-defined function called at program startup. This function must have external linkage (it is possible for a translation unit to declare a function or object, called `main`, with internal linkage). There is no requirement that the function appear in a source file having a specific name.

Other Languages

A Java Machine starts execution by invoking the method main of some specified class, . . .

In Fortran and Pascal the function called on startup is defined using the keyword **PROGRAM**.

Common Implementations

Implementations use a variety of techniques to start executing a program image. These usually involve executing some internal, implementation-specific, function before `main` is called. The association between this internal function and `main` is usually made at link-time by having the internal function make a call to `main`, which is resolved by the linker in the same way as other function calls.

Coding Guidelines

Having the function `main` defined in a source file whose name is the same as the name of the executable program is a commonly seen convention.

The implementation declares no prototype for this function. 163

Commentary

There is no declaration of `main` in any system header, or internally within the translator. For this reason translators rarely check that the definition of `main` follows one of the two specifications given for it.

Common Implementations

A function named `main` is usually treated like any other developer-defined function.

Coding Guidelines

Because the implementation is not required to define a prototype for `main` no consistency checks need be performed against the definition of `main` written by the developer.

It shall be defined with a return type of **int** and with no parameters: 164

```
int main(void) { /* ... */ }
```

Commentary

Many lazy developers use the form:

```
1  main() { /* ...  */ }
```

C99 does not allow declaration specifiers to be omitted; a return type must now be specified. 1379 declaration at least one type specifier

Usage

There was not a sufficiently large number of instances of `main` in the `.c` files to provide a reliable measure of the different ways this function is declared.

---

165 or with two parameters (referred to here as **argc** and **argv**, though any names may be used, as they are local to the function in which they are declared): main prototype

```
int main(int argc, char *argv[]) { /* ... */ }
```

Commentary

The parameters of `main` are treated just like the parameters in any other developer-defined function.

Other Languages

. . . , passing it a single argument, which is an array of strings. Java

```
class Test {
   public static void main(String[] args) {
      /* ... */
      }
   }
```

Coding Guidelines

The identifiers `argc` and `argv` are commonly used by developers as the names of the parameters to `main` (the standard does not require this usage).

---

166 or equivalent;[9)]

Commentary

Many developers use the form:

```
1  int main(int argc, char **argv)
```

on the basis that arrays are converted to pointers to their element type.

C++

The C++ Standard gives no such explicit permission.

---

167 or in some other implementation-defined manner.

Commentary

The Committee recognized that there is additional information that a host might need to pass to `main` on startup, but they did not want to mandate anything specific. The following invocations are often seen in source code:

```
1  void main()
```

and

```
1  int main(argc, argv, envp)
```

C90

Support for this latitude is new in C99.

C++

The C++ Standard explicitly gives permission for an implementation to define this function using different parameter types, but it specifies that the return type is **int**.

3.6.1p2 *It shall have a return type of* ***int****, but otherwise its type is implementation-defined.*

. . .

[Note: it is recommended that any further (optional) parameters be added after `argv`*.]*

Common Implementations

In a wide character hosted environment, `main` may be defined as:

```
int main (int argc, wchar_t *argv[])
```

Some implementations[600] support a definition of the form:

```
int main (int argc, char *argv[], char *env[])
```

where the third parameter is an array environment strings such as `"PATH=/home/derek"`.

The POSIX `execv` series of functions pass this kind of additional information to the invoked program. However, `main` is still defined to take two parameters for these functions; the environment information is assigned to an external object `extern char **environ`.

Coding Guidelines

Programs that need to make use of the implementation-defined functionality will obviously define `main` appropriately. If it is necessary to access environmental information, it is best done through use of implementation-supported functionality.

If they are declared, the parameters to the **main** function shall obey the following constraints: 168

Commentary

This is a list of requirements on implementations. It lists the properties that the values of `argc` and `argv` can be relied on, by a program, to possess. The standard does not define any constraints on the values of any, implementation-defined, additional parameters to `main`.

Common Implementations

The constraints have proved to be sufficiently broad to be implementable on a wide range of hosts.

Coding Guidelines

These constraints are the minimum specification that can be relied on to be supported by an implementation.

— The value of **argc** shall be nonnegative. 169

Commentary

The value of `argc`, as set by the implementation on startup is nonnegative. The parameter has type **int** and could be set to a negative value within the program.

Common Implementations

Most hosts have a limit on the maximum number of characters that may be passed to a program on startup (via, for instance, the command line). This limit is also likely to be a maximum upper limit on the value of `argc`.

Coding Guidelines

Knowing that `argc` is always set to a nonnegative value, a developer reading the code might expect it to always have a nonnegative value. Assigning a negative value to `argc` as a flag to indicate a special condition is sometimes seen. The extent to which this might be considered poor practice because of violated developer expectations is discussed elsewhere.

1352 object role

---

170 — **`argv[argc]`** shall be a null pointer.

Commentary

Some algorithmic styles prefer to decrement a count of the remaining elements. Some prefer to walk a data structure until a null pointer is encountered. The standard supports both styles.

---

171 — If the value of **`argc`** is greater than zero, the array members **`argv[0]`** through **`argv[argc-1]`** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup.

argv values

Commentary

The storage occupied by the strings has static storage-duration. This wording could be interpreted to imply an ordering between the contents of the host environment and the contents of `argv`. What the user typed on a command line may not be what gets passed to the program. The host environment is likely to process the arguments first, possibly performing such operations as expanding wildcards and environment variables.

455 static storage duration

Common Implementations

A white-space character is usually used to delimit program parameters. Many hosts offer some form of quoting mechanism to allow any representable character, in the host environment, to be passed as a value to `argv` (including character sequences that contain white space). The behavior most often encountered is that the elements of `argv`, in increasing subscript order, correspond to the white-space delimited character sequences appearing on the command line in left-to-right order.

POSIX defines `_POSIX_ARG_MAX`, in **`<limits.h>`**, as "The length of the arguments for one of the exec functions, in bytes, including environment data." and requires that it have a value of at least 4,096. Under MS-DOS the command processor, `command.com`, reads a value on startup that specifies the maximum amount of the space to use for its environment (where command line information is held).

Coding Guidelines

Most implementations have a limit on the total number of characters, over all the strings, that can be passed via `argv`. There is a possibility that one of the character sequences in one of the strings pointed to by `argv` will be truncated.

For GUI-based applications the values passed to `argv` may be hidden from the user of the application.

Many hosts offer some form of quoting mechanism to allow any typeable character, in that environment, to be passed to a program. If `argv` is used, a program may need to handle strings containing characters that are not contained in the basic execution character set.

---

172 The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment.

main parameters intent

Commentary

The `argc`/`argv` mechanism provides a simple-to-use, from both the applications' users' point of view and the developer's, method of passing information (usually frequently changing) to a program. The alternative being to use a file, read by the application, which would need to be created or edited every time options changed. This is not to say that an implementation might not choose to obtain the information by reading from a file (e.g., in a GUI environment where there is no command line).

C++

The C++ Standard does not specify any intent behind its support for this functionality.

Common Implementations

If the program was invoked from the command line, the arguments are usually the sequence of characters appearing after the name of the program on the line containing the program image name. The order of the strings appearing in successive elements of the `argv` array is the same order as they appeared on the command line and the space character is treated as the delimiter.

In GUI environments more information is likely to be contained in configuration files associated with the program to be executed. Many GUIs implement click, or drag-and-drop, by invoking a program with `argv[1]` denoting the name of the clicked, or dropped-on, file and `argv[2]` denoting the name of the dropped file, or some similar convention.

Coding Guidelines

Making use of `argv` necessarily relies on implementation-defined behavior. Where the information is obtained, and the ordering of strings in `argv`, are just some of the issues that need to be considered.

---

argv lowercase

If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase. 173

Commentary

This choice reflects C's origins in Unix environments. Some older host environments convert, by default, all command line character input into a single case. A commonly occurring program argument is the name of a file. Unix filenames tend to be in lowercase and the file system is case-sensitive. Having an application accept non-filename arguments in lowercase has a lower cost than insisting that filenames be in uppercase.

C++

The C++ Standard is silent on this issue.

Common Implementations

In some mainframe environments, the convention is often to use uppercase letters. Although entering lowercase letters is difficult, it has usually proven possible to implement.

Coding Guidelines

Most modern hosts support the use of both upper- and lowercase characters. The extent to which a program needs to take account of those cases where they might not both be available will depend on the information passed via `argv` and the likelihood that the program will need to be ported to such a host.

---

argv program name

— If the value of **`argc`** is greater than zero, the string pointed to by **`argv[0]`** represents the *program name*; 174

Commentary

What is the program name? It is usually thought of as the sequence of characters used to invoke the program, which in turn relates in some way to the program image (perhaps the name of a file).

Common Implementations

Under MS-DOS typing, *abc* on the command line causes the program `abc.exe` to be executed. In most implementations, the value of `argv[0]` in this case is *abc.exe* even though the extension `.exe` may not have appeared on the command line.

Unix based environments use the character sequence typed on the command line as the name of the program. This does not usually include the search path along which the program was found. For symbolic links the name of the symbolic link is usually used, not the name of the file linked to.

Coding Guidelines

Some programs use the name under which they were invoked to modify their behavior, for instance, the GNU file compression program is called `gzip`. Running the same program, having renamed it to `gunzip`, causes it

to uncompress files. Running the same program, renamed to `gzcat`, causes it to read its input from `stdin` and send its output to `stdout`. The standard distribution of this software has a single program image and various symbolic links to it, each with different names.

The extent to which the behavior of a program depends on the value of `argv[0]` is outside the scope of these coding guidelines.

175 **`argv[0][0]`** shall be the null character if the program name is not available from the host environment.

Commentary

Not all host environments can provide information on the name of the currently executing program.

Common Implementations

Most hosts can provide the name of the program via `argv[0]`.

176 If the value of **`argc`** is greater than one, the strings pointed to by **`argv[1]`** through **`argv[argc-1]`** represent the *program parameters*.

program parameters

Commentary

This defines the term *program parameters*. It does not require that the value of the parameters come from the command line, although this is the background to the terminology.

Example

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5  printf("The program parameters were:\n");
6
7  for (int a_index = 0; a_index < argc; a_index++)
8     printf("%s\n", argv[a_index]);
9  }
```

177— The parameters **`argc`** and **`argv`** and the strings pointed to by the **`argv`** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

Commentary

Although the strings pointed to by the `argv` array are required to be modifiable, the array itself is not required to be modifiable. Calling `main` recursively (permitted in C90, but not in C99) does not cause the original values to be assigned to those parameters.

C++

The C++ Standard is silent on this issue.

Common Implementations

The addresses of the storage used to hold the `argv` strings may be disjoint from the stack and heap storage areas assigned to a program on startup. Some implementations choose to place the program parameter strings in an area of storage specifically reserved, by the host environment, for programs' parameters.

5.1.2.2.2 Program execution

178 In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7).

Commentary

There are no library subsets; C is a single implementation conformance level standard. However, although functions of the specified name may be present to link against, in translation phase 8, the functionality provided by an implementation's library may vary (including doing nothing and effectively being optional).

Some library functionality was known to be difficult, if not impossible, to implement on certain host environments. In these cases the library functions have minimal required functionality; for instance, signal handling.

implementation extensions 95

The C Standard, unlike POSIX, does not prohibit the use of functions, macros, type definitions, and objects from other standards, but such libraries must not change the behavior of any of the C-defined library functionality.

Other Languages

Cobol, SQL, and Pascal are multi-conformance level standards. They contain additional language constructs at each level. A program is defined to be at the level of the highest level construct it uses.

---

footnote 9

9) Thus, **int** can be replaced by a typedef name defined as **int**, or the type of **argv** can be written as **char** ** **argv**, and so on. 179

Commentary

typedef is synonym 1633

The return type might also be specified as **signed int**. A typedef name in C is a synonym.

Some implementations choose to process a function called `main` differently from other function definitions (because of its special status as the function called on program startup). This wording makes it clear that the recognition of `main`, its return type, and arguments cannot be based on a textual match. Semantic analysis is required.

C++

The C++ Standard does not make this observation.

Coding Guidelines

Replacing one or more of these types by a typedef name suggests that the underlying type may change at some point. Gratuitous use of a typedef name is not worth a guideline recommending against it. Other uses are for a purpose and there is no reason for them to be the subject of a guideline.

5.1.2.2.3 Program termination

---

main return equivalent to

If the return type of the **main** function is a type compatible with **int**, a return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument;[10)] 180

Commentary

The call to `exit` will invoke any functions that have been registered with the `atexit` function. This behavior is not the same as a return from `main` simply returning control to whatever startup function called it, which in turn calls `exit`. After `main` returns, any objects defined in it with automatic storage will no longer have storage reserved for them, rendering any access (i.e., in the functions registered with `atexit`) to them as undefined behavior.

C90

Support for a return type of `main` other than **int** is new in C99.

C++

The C++ wording is essentially the same as C90.

Common Implementations

The value returned is often used as the termination status of the executed program in the host environment. Many host environments provide a mechanism for testing this status immediately after program termination.

Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  signed char *gp;
5
6  void clean_up(void)
7  {
8  /*
9   * The lifetime of the object pointed to by gp
10  * has terminated by the time we get here.
11  */
12 printf("Value=%c\n", *gp);
13 }
14
15 int main(void)
16 {
17 signed char lc;
18
19 if (atexit(clean_up) != 0)
20    printf("Ho hum\n");
21
22 gp=&lc;
23
24 return 0;
25 }
```

181 reaching the **}** that terminates the **main** function returns a value of 0.

Commentary

The standard finally having to bow to sloppy existing practices.

C90

This requirement is new in C99.

If the main function executes a return that specifies no value, the termination status returned to the host environment is undefined.

Common Implementations

Many implementations followed this C99 specification in C90, prior to it being explicitly specified in C99.

A translator can choose to special case functions defined with the name `main` or simply provide, where necessary, an implicit return of 0 for all functions returning type **int**.

182 If the return type is not compatible with **int**, the termination status returned to the host environment is unspecified.

main termination status unspecified

Commentary

The return type can only be incompatible with **int** if `main` has been defined to return a different, incompatible, type. If `main` has a return type of **void**, an implementation may choose not to return any value. In this case, the standard does not guarantee that a call to the `exit` library function will return the requested status to the host environment.

C90

Support `main` returning a type that is not compatible with **int** is new in C99.

C++

3.6.1p2 *It shall have a return type of* ***int****, . . .*

Like C90, C++ does not support `main` having any return type other than **`int`**.

Forward references: definition of terms (7.1.1), the **`exit`** function (7.20.4.3). 183

5.1.2.3 Program execution

program execution
abstract machine
C

The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant. 184

Commentary

The properties of this abstract machine are never fully defined in the standard. The term can be thought of as a conceptual model used to discuss C, not as a formal definition for a full-blown specification. Several research projects have produced formal specifications of substantial parts of Standard C (using operational semantics based on evolving algebras,[528] structural operational semantics,[1016] and denotational semantics[1052]). These have investigated the language only (no preprocessor and a subset of the library).

The semantic descriptions also ignore other possible ways of modifying values in storage; for instance, by irradiating the processor or storage chips, forming the computing platform, with nuclear or electromagnetic radiation.[509]

Other Languages

The definition of most computer languages is written in a form of stylized English. The Modula-2 Standard[646] was defined using a formal definition language[647] (which in turn is defined in terms of Zermelo-Fraenkel set theory; which in turn is based on nine axioms, {although only four of these have been proved to be consistent and independent}) and English. Neither language was given priority over the other. Any difference in meaning between the two formalisms has to be decided by a discussing the intended behavior, not by giving one set of wording preference over the other. The original definition of Lisp was written in terms of a subset of itself and the Prolog Standard[644] provides (in an informative annex) a formal semantics written in a subset of Prolog. A Model Implementation (an implementation that exactly implements all of the requirements of a language standard, irrespective of performance issues) was created for Pascal.[1453]

Common Implementations

Most vendors regard optimizations as being very important and sometimes invest more effort on performing them than doing anything else.

A Model Implementation was produced for C.[681] This implementation aimed to mimic the behavior of the abstract machine and diagnose all implementation-defined, unspecified and undefined behaviors (as well as the usual constraint and syntax violations). It generated code for an abstract machine whose instruction set had a one-to-one mapping to C operations, the idea being that such a one-to-one mapping helped simplify code generation and reduce possible bugs. The interpreter for this abstract machine did all the pointer checks and uninitialized object checks that caused undefined behavior.

At the time of this writing there is one formally verified compiler[127] for a substantial subset of C.

Coding Guidelines

Many developers spend a relatively large amount of time trying to optimize their code. The extent to which their efforts have any effect is often marginal. There are a lot of misconceptions about how to write efficient C. By far the biggest improvements in performance come from design and algorithmic optimization. Trying to second guess what machine code a translator will generate requires a great deal of knowledge on the target architecture and the code-generation techniques used by an implementation, skills that very few developers

have. The only effective way to tune at the statement level is by looking at the generated machine code. Changes based on ideas about what the translator *should do* are often wrong.

While the benefits of using formal methods when writing software are often promoted by their followers, the associated costs[424] and lack of any evidence that the benefits can be realized in practice[529] are rarely mentioned.

---

185 Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,[11] which are changes in the state of the execution environment.

side effect

Commentary

An operation, in a C program, that does not generate a side effect is, at the very least, considered to be of no practical interest.

190 redundant code

The state of the execution environment includes information on the current flow of control (nested function calls and the current statement being executed). This means that the control expression in, for instance, an `if` statement has the side effect of selecting the flow of control. Accessing a volatile object is not guaranteed to change the state of the execution environment, but a translator must act as-if it does. The C library treats all I/O operations as occurring on files, and therefore generating a side effect. The C++ Standard is more explicit in stating, 1.9, " . . . calling a library I/O function, . . . ".

Other Languages

Some languages, usually functional, have been specifically designed to be side effect free. The advantage of being side effect free is that it significantly simplifies the task of mathematically proving various properties about programs.

Common Implementations

Calling a function, or one of the memory allocation functions, also changes the state of the execution environment. This state is part of the housekeeping performed by an implementation and does not normally cause an external effect. Such operations only become noticeable if there are insufficient resources to satisfy them.

Coding Guidelines

C is what is known as an imperative language. The design of C programs (and nearly every commercially written program, irrespective of language used) is generally based on using side effects to implement the desired functionality. Coding guideline documents that recommend against the use of side effects (which they do with surprising regularity) are rather missing the point. Developers need to ensure that side effects occur in a predictable order. The relevant issues are discussed elsewhere.

944 expression order of evaluation

Comprehending source code requires readers to deduce the changes it makes to the state of the abstract machine. This process requires that readers remember, and later recall, a sequence of abstract machine states. Some of the issues involved in organizing changes of machine state into a meaningful pattern are discussed elsewhere, as are issues of locally minimizing the number of changes of state that need to be remembered.

1707 statement syntax
1046 postfix operator constraint

---

186 Evaluation of an expression may produce side effects.

Commentary

An expression may also be evaluated for its result value. This value is used to decide the flow of control, or returned as the value of a function call.

940 expressions

Operations on objects having floating-point type may also set status flags, which is a change of state of the execution environment.

196 footnote 11

Common Implementations

Some optimizers invoke the as-if rule to delay the updating of storage after the evaluation of an expression. The storage may not be necessary if the value can be kept in a register and accessed their for all subsequent accesses (or at least until it is modified again).

Coding Guidelines

One of the most common reader activities when performing source code comprehension is the analysis of the consequences of side effects in the evaluation of expressions. An expression that does not produce side effects, when evaluated, is redundant. The issue of redundant code is discussed elsewhere.

redundant code 190

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  glob+4;          /* No side effect. */
6
7  if (glob+4 == 0) /* Evaluation selects the flow of control. */
8     glob--;       /* A side effect. */
9  }
```

---

sequence points

At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. 187

Commentary

This defines the term *sequence points*. Sequence points are points of stability. At these points there are no outstanding modifications to objects waiting to be completed. The ordering of sequence points during program execution is not always guaranteed. The freedom given to translators in evaluating binary operands leaves open the possibility of there being more than one possible ordering of sequence points. The following are some of the more important orderings implied by sequence points:

sequence points 187

object initializer evaluated when 1711
statement executed in sequence 1709
comma operator syntax 1313

- Declaration evaluation order.
- Statement execution order.
- A left then right evaluation ordering of operands of some binary operators.

In some situations the specification given in the standard has been found to be open to more than one interpretation. Various C Committee members are working on a more formal specification of the semantics of expressions. Several notations have been proposed (e.g., using sets or trees); the intent is to select the one that developers will find the simplest to use. Whether the final document will be published as a TR or in some other form has not yet been decided. The working papers are available via the WG14 Web site.

WG14/N899 *To define a formalism for the semantics of expressions in the C language (as defined in ISO/IEC 9899:1999) to enable users of the Standard to determine unambiguously what expressions do or do not conform to the language and their level of conformance.*

An example of the nontrivial issues that surround the analysis of expressions that contain more than one sequence point is provided by DR #087. In the following code, is the behavior for the expressions in lines A, B, and C defined?

```
1  static int glob;
2
3  int f(int p)
4  {
5  glob = p;
6  return 0;
7  }
```

```
8
9   void DR_087(void)
10  {
11  int loc;
12
13  loc = (glob = 1) + (glob = 2);                 /* Line C */
14  loc = (10, glob = 1, 20) + (30, glob = 2, 40); /* Line A */
15  loc = (10, f (1), 20) + (30, f (2), 40);       /* Line B */
16  }
```

In *Line C* there are two assignments to `glob` between two sequence points. The behavior is undefined. In *Line A* the assignments to `glob` are bracketed by sequence points. It might be thought that there is no possibility that the two assignments could both occur between the same pair of sequence points. However, there is no requirement that an operator be performed immediately after one or more of its operands is evaluated. One of the evaluation orders an implementation could choose is:

```
evaluate 10
sequence point
evaluate 30
sequence point
evaluate glob=1
evaluate glob=2
sequence point
evaluate 20
sequence point
evaluate 40
perform addition
perform assignment
sequence point
```

Just like the case in *Line C* there are two assignments to `glob` between two sequence points. The behavior is undefined.

function call interleave

In *Line B* there are two calls to the function `f`; however, calls to functions cannot be interleaved (although this requirement appears in a number of committee papers dealing with sequence point and is part of the question in DR #087, it has never been stated in any normative document). This means that the two assignments to `glob`, which occur in the two calls to `f`, can never occur between the same pair of sequence points.

Other Languages

Most languages do not provide the operators, available in C, for modifying objects within an expression. However, most languages do allow function calls within an expression and these can modify objects with file scope or indirectly through parameters. Some language definitions specify that changing the value of a variable in an expression, while it is being evaluated (i.e., via function call), always results in undefined behavior (or some such term). Very few languages discuss the interaction of side effects and sequence points (if any such concept is defined).

Common Implementations

The as-if rule can be invoked to allow code motion across sequence points. However, implementations have to ensure that the behavior remains unchanged (in terms of external output or behavior, if not execution-time performance). The only way for a developer to find out if an implementation is using the as-if rule to reorder side effects around sequence points is to examine the generated code. Many implementations do provide an option to list the machine code generated.

Most C expressions are very simple, offering little opportunity for clever optimization in themselves. Obtaining high-quality code requires finding similarities within expressions occurring in sequences of statements, that is across sequence points. Value numbering is a commonly used technique.[160]

Coding Guidelines

The sequence point definition suggests, on first glance, a well-defined ordering of events. In most cases the ordering is predictable, but in a few cases the ordering of sequence points is not unique. Some of the constructs that are sequence points can themselves be evaluated in different orders. The two sequence points that cannot have multiple orderings are the `;` punctuator at the end of a statement and initialization of block scope definitions, and the evaluation of a full expression that is also a controlling expression. The possible orderings of these sequence points is fully specified by the standard and cannot vary between implementations.

All other sequence points can occur within a subexpression that is the operand of a larger expression. A sequence point may guarantee an evaluation order within a subexpression. However, there might not be any guarantee that the subexpression is always evaluated at the same point in the evaluation of its containing full expression. For instance in:

```
1  f(c++) + g(c -= 3);
```

the sequence point before a `( )` operator is invoked guarantees that either `c` will be incremented before `f` is called, or that `c` will be incremented before `g` is called. The standard does not specify which function should be called first, so there are two possible orderings of sequence points. In:

```
1  (x--, p = x+y) + (y++, q = x+y)
```

the sequence point on the comma operator guarantees that its left operand will be evaluated and all side effects will take place before the right operand is evaluated. But nothing is said about which operand of the plus operator is evaluated first.

Cg 187.1 The value of a full expression shall not depend on the order of evaluation of any sequence points it contains.

Cg 187.2 Any types declared in a declaration shall not depend on the order of evaluation of any expressions it contains.

object 1711 initializer evaluated when

The corresponding coding guideline issues for initializers are discussed elsewhere.

Example

```
1  extern int glob;
2  extern int g(int);
3
4  void f(void)
5  {
6  int loc = g(glob = 3) + g(glob = 4);
7  }
```

(A summary of the sequence points is given in annex C.) 188

---

expression evaluation abstract machine

In the abstract machine, all expressions are evaluated as specified by the semantics. 189

Commentary

The abstract machine has no concern for the size of the program being executed, or its execution-time performance.

C++

The C++ Standard specifies no such requirement.

Common Implementations

When translating programs for use with a symbolic debugger, most implementations tend to closely follow the requirements of the abstract machine. This ensures that the values of objects, in memory, closely follow their evaluations in the source code. This simplifies things considerably for developers, who might be having enough trouble following the behavior of a program, and don't need the additional confusion of objects receiving values at different times to those specified in the source.

This seemingly innocuous requirement prohibits implementations from performing expression rewriting. 943 expression grouping

Coding Guidelines

Trying to second-guess how an implementation might process a given construct is generally a fruitless exercise. Assuming that an implementation behaves like the abstract machine is a useful starting point that does not often need to be improved on.

Example

In the following an implementation may spot that the subexpression `x*y` occurs in two places and the values of `x` and `y` are not modified between these two points. The extent to which this information can be used will depend on the number of registers available for holding temporary values, the relative cost of evaluating `x*y`, and the priority given to this optimization opportunity relative to other opportunities within the function `f`.

```
1  extern int x, y;
2
3  void f(void)
4  {
5  int a,
6      b;
7
8  a = x * y;
9  /*
10  * Code that does not change the values of x and y.
11  */
12 b = x * y;
13 }
```

---

190 An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

expression need not evaluate part of

Commentary

This statement forms part of the as-if rule. 122 as-if rule

In the following the value is used, but it is also known at translation time. To answer the question of whether there are any needed side effects requires varying degrees of source analysis sophistication:

```
1  extern int g(void);
2  extern int glob;
3
4  void f(void)
5  {
6  int loc;
7
8  loc = 0 * g();        /* Result always zero, but could have needed side effect. */
9  loc = 0 * (glob + 4); /* Result always zero, no side effects. */
10 }
```

The value assigned to `loc` is known in both cases. In the first case the call to the function `g` must occur if it causes needed (the program output depends on them) side effects. In the second case `(glob + 4)` only need be evaluated if the object `glob` is declared with the volatile storage qualifier. The cases differ only in the easy with which a translator can deduce that a needed side effect occurs.

Those cases where there may be no needed side effect apply to binary operators. Operator/operand pairs where evaluations need not occur and consequently there may be no needed side effects include:

```
0 * (expr)
0 & (expr) always 0
0xffff | (expr) always 0xffff (or appropriate sized constant)
(expr) - (expr) always 0
(expr) / (expr) always 1
```

The expression `expr / expr` might cause a side effect because of the undefined behavior that occurs when `expr` is 0. An implementation is at liberty to deliver the result `1` (if the original code is simplified), raise an exception (if that is what the host processor does for this case), or carry out some other action.

There are also cases where a constant operand does not affect the result of an expression, but the other operand still needs to be evaluated. For instance: adding zero, shifting by zero bits positions, the **%** operator with a right-hand side of one, or multiplying by one. In some cases the cast operator can have no effect. In the function:

```
1  static long glob;
2
3  int f(void)
4  {
5  return (int)glob;
6  }
```

the cast to **int** would be redundant if **int** and **long** had the same representation.

Common Implementations

The extent to which implementations perform the analysis necessary to deduce that an expression, or part of it, is not used varies enormously. The simpler optimizing translators tend to spend their time producing the best code from the source code on a statement-by-statement basis. These tend to treat all parts of an expression as being needed. More sophisticated optimizers analyze blocks of statements, trying locate common subexpressions, CSEs, and performing some level of flow analysis. This level of optimization is willing to believe that certain kinds of subexpressions may be redundant.

A translator that generates very high performance code is of no use if the final behavior is incorrect. The savings to be made from removing some redundant subexpressions are sometimes not worth the risk of getting it wrong. The translator that optimizes:

```
1  #define A_OFFSET(x) (1+(x))
2
3  extern int p, q;
4  static volatile int r;
5
6  void f(void)
7  {
8  p=A_OFFSET(q) - r / r;
9  }
```

into `p=q` would either be foolhardy or perform a great deal of analysis of the program and its environment.

A translator can optimize those cases where the operands always have known values. However, studies have shown[1164] that a significant number of these redundant operations still occur during program execution (because the calculated values of operands happen to be zero or one).

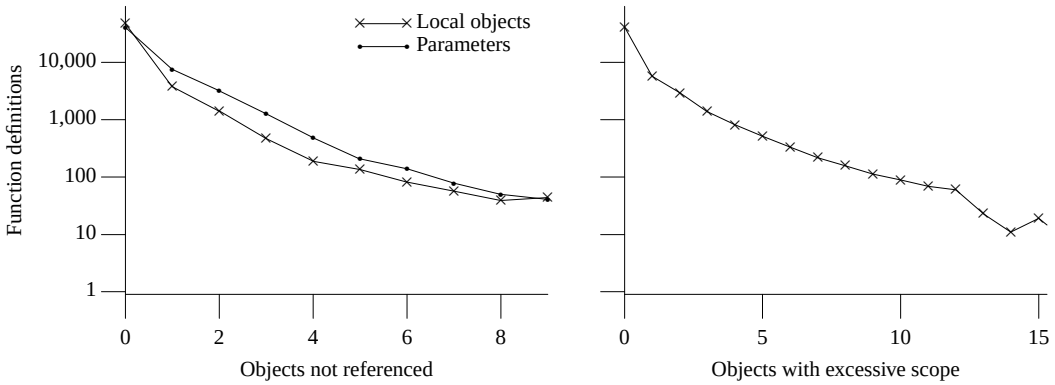


Figure 190.1: Number of parameters or locally defined objects that are not referenced within a function definition (left graph); number of objects declared in a scope that is wider than that needed for any of the references to them within a function definition (right graph). Based on the translated form of this book's benchmark programs.

In some cases code is only redundant under certain conditions. For instance, its evaluation may only be necessary along a subset of the control flow paths that pass through it. So called *partial redundancy elimination* algorithms involve code restructuring to remove this redundancy. Ensuring that this code motion and duplication does not result in excessive code bloat requires information on the execution-time characteristics of the program.[131] partial redundancy elimination

There have been proposals[954] for detecting, in hardware, common subexpression evaluations during program execution and reusing previously computed (and stored in a special table) results.

Coding Guidelines

There are cases where, technically, part of an expression has no effect on the final result but where from the human point of view the operation needs to exist in the source code— including the following:

- Source code that is translated and forms part of the program image, but is never executed, is known as *dead code*. There is no commonly used equivalent term denoting objects that are defined but never referenced (see Figure 190.1). dead code
- Source code that is translated and forms part of the program image, whose execution does not affect the external appearance or output of a program, is known as *redundant code*. Declaring objects to have a scope wider than is necessary (e.g., an object declared in the outermost block that is only referenced within one nested block— Figure 190.1) is not generally considered to be giving them a redundant scope. redundant code
- Source code that appears within conditional inclusion directives (e.g., **`#if`**/**`#endif`**) is not classified as dead code. It may not be translated to form part of a program image in some cases, but in other cases it may be.

The issue of duplicate code is discussed elsewhere. Dead, or redundant, code can increase costs in a number of ways, including the following: 1821 duplicate code

- *Additional maintenance effort.* It is source code that developers do not need to read. They may even make changes to it in the mistaken belief that it affects program output.
- *Consume host processor resources.* For instance, in the case of unused storage, execution performance can be affected through its effect on cache behavior. 0 cache

This kind of code can exist for a number of reasons, including:

- Defensive programming:
  - providing a default label even when it can be shown that all possible switch values are covered by labels
  - guideline recommendations that require a return at the end of a function, even though flow analysis shows it is never reached
- Conditional compilation based on translation-time constants in the conditional expressions of **`if`** statements, instead of using **`#if`**/**`#endif`**.
- Coding error.
- Using symbolic representations. For instance, a symbolic name, representing a numeric value, appearing as an operand, or the typedef name used in a cast operation that happens to denote the same C type as operand.

symbolic 822 name cast-1133 expression syntax

- Design oversight:
  - an **`if`** statement that can never be true, or is always false, based on characteristics of the application
- Cut-and-paste of existing code rather than creating a new function.
- Specification changes:
  - conditions that could previously have been true become impossible to satisfy
- At the highest level, code may only be executed when certain hardware is available (e.g., joystick).
- A developer is aware of the situation but does not consider it worth investing effort to remove the code.

Locating all dead code within a program is technically a very difficult problem. The most commonly seen levels of analysis operate at the function and statement level. Automatic tools provide a partial solution to dealing with the complexities of detecting dead code. Such tools can operate at differing levels of sophistication:

- analyze each expression in isolation:
  - detects expressions that do not contain any side effects
- simple flow analysis within a function:
  - detects redundant statements
- symbolic flow analysis within a function, called intraprocedural analysis:
  - detects **`if`**/**`while`**/**`for`** never/always executed
- flow analysis, including information across function calls, called interprocedural analysis:
  - the additional information allows a more thorough job to be done
- complete program analysis:
  - Many linkers do not include function definitions that are never referenced in the program image. The extent to which objects that are not references are included in a program image varies.

linkers 140 program 141 image

In practice, given current tool limitations, and theoretical problems associated with incomplete information (i.e., the set of values input to a program), it is unlikely that all dead code be detected.

Locating redundant code requires additional effort since it is necessary to show that particular side effects, which are performed, have no effect on the external behavior. An example of a case that is often flagged in an object defined and assigned a value within a function, but which is never referenced.

A study by Xie and Engler[1486] attempted to find a correlation between redundant code and faults that existed in that code.

The extent to which the cost of including dead and redundant code in a program outweighs the cost of removing it is a management decision. It is not unusual to find that 30%, or more, of the functions in an application, which has evolved over many years, to be uncalled (i.e., they are dead code)[1355] (see Table 1821.3). It is also common to find objects that are defined and never referenced, and **#include**s that are unnecessary (see Figure 1896.2). Dead code within functions that are called does occur, but not in significant quantities. In the case of duplicate code 5.9% of the lines of `gcc` have been found to be duplicate,[370] a study[102] of a 400 K C source line commercial application found an average duplication rate of 12.7%.

1740 controlling expression if statement

Example

```
#include <stdio.h>

extern int get_status(void);
static unsigned int glob_2;

void f(void)  /* Never called. */
{
glob_1++;
}

int main(void)
{
int local_1 = 11;

glob_1 = get_status();

if (0)
   local_1--; /* Never executed. */
if (local_1 == 12)
   glob_1--;  /* Never executed. */

if (glob_1 == 0)
   glob_1 = 1;
if (glob_1 > 0) /* Always true, but needs logical deduction. */
   printf("Hello world\n");
else
   printf("Illogical world\n"); /* Never executed. */
}
```

191 When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on.

signal interrupt abstract machine processing

Commentary

C has no restrictions on when a signal can be raised. If it occurs via a call to the `raise` function there will have been a sequence point before the call. If the signal is raised as a result of some external mechanism, for instance a timer interrupt or the instruction being executed raising some signal, then the developer has no control over when it occurs.

Assuming an average of 3 machine code instructions for every source code statement gives a 66% chance (ignoring the issue of sequence points within the statement) of a C statement being partially executed when a signal occurs.

C++

1.9p9 *When the processing of the abstract machine is interrupted by receipt of a signal, the value of any objects with type other than* `volatile sig_atomic_t` *are unspecified, and the value of any object not of* `volatile sig_atomic_t` *that is modified by the handler becomes undefined.*

This additional wording closely follows that given in the description of the `signal` function in the library clause of the C Standard.

Common Implementations

processor 0 pipeline

To improve performance modern processors hold a pipeline of instructions at various stages of completion. On some such processors, a signal is not processed until the pipeline has completed executing all of the instructions it contained when the signal was first received. For such processors it might not even be possible to associate a signal with a given, previous sequence point. Roberts[1172] and Gehani[477] describe various implementation and application issues associated with signal handling.

Coding Guidelines

Because a signal handler does not have any knowledge of where a signal is likely to be raised, it cannot assume anything about the values of objects it references. Through knowledge of the source code and the host environment, a developer may be able to narrow down the locations where certain signals might be expected to be raised. Because it is not possible to say anything with certainty and the difficultly of performing automatic checking a guideline review recommendation is made.

Rev 191.1

When designing and implementing programs that operate in the presence of signals no assumptions shall be made about the values of objects modified by statements in the vicinity of the flow of control where a caught signal could be raised.

Example

In the following:

```
1  #include <signal.h>
2  #include <stdio.h>
3
4  extern volatile sig_atomic_t flag;
5  extern double value,
6                A, B;
7
8  void sig_handler(int sig_number)
9  {
10 if (flag == 1)
11    printf("Need not be after A / B\n");
12 }
13
14 void f(void)
15 {
16 signal(SIGFPE, sig_handler);
17
18 flag = 0;
19 value = A / B;
20 flag = 1;
21 }
```

The instructions to assign 1 to f might already be in the processor pipeline when the floating-point divide raises a signal. The function sig_handler cannot assume anything about the value of flag even though it has sig_atomic_t type.

The only solution to the pipeline problem is to insert sufficient statements between the floating-point operation and the assignment to flag to ensure that the assignment instructions are not in the pipeline when the signal is raised. Even this is not a guaranteed solution because the translator might deduce (incorrectly in this case) that the machine code for these intervening statements could be moved to some other point in the flow of execution.

---

192 Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

modified objects received correct value

Commentary

It is not even possible to assume that objects will have one of two possible values (their previous or new value). In the two assignment:

```
A = 0x00ff;
/* Some code. */
A = 0xff00;
```

if a signal occurs during the evaluation of the second expression statement, the possible values the object A can take include 0x0000, 0x00ff, 0xffff, or 0xff00. These alternatives can occur if the underlying processor transfers multibyte values to storage one byte at a time.

The requirements on the Library typedef sig_atomic_t ensure that objects defined with this type will be accessed as an atomic operation. In the preceding example, for a processor with the byte at a time characteristics, the implementation would probably choose to use a character type in the definition of sig_atomic_t.

C++

The C++ Standard does not make this observation.

Common Implementations

It is rare for processors to support the handling of an interrupt after the partial execution of an instruction. This behavior is sometimes seen for instructions that move large amounts of data; for instance, string-handling instructions, where the processor may loop until some condition is met, can potentially take a very (relatively) long time to execute. Processors generally want to respond to an interrupt within a specified time limit.

Coding Guidelines

The myriad of possible problems that can occur as a result of an object not receiving its correct value, because of a signal being raised, are to0 diverse (and specialized) to be covered in a few guidelines.

---

193 The least requirements on a conforming implementation are:

implementation least requirements

Commentary

The *least requirements* listed provide the basis for ensuring that the output appears in the order required by the author of the code. However, the standard says nothing about how quickly it will appear after the statement performing the operation has finished executing. The only way of telling whether an implementation has ordered the operations that occurred during the execution of a program according to these requirements is to examine the output produced.

The workings of the abstract machine between the user issuing the command to execute a program image and the termination of that program is unknown to the outside world. For instance, a translator may recognize that a particular sequence of source code will output a given number of digits of the value of π. It may then translate the source into a program that prints Ascii characters from an internal table, to the desired precision, instead of executing the algorithm to calculate the digits contained in the program's source.

184 abstract machine C
205 semantics stringent correspondence

Some naive benchmark programs contain loops that have no effect on the output of the program (usually for some timing purpose). Users of such benchmarks continue to be confused by the timing results obtained from using an optimizing translator that deduces that it does not need to generate any executable code for these loops.

Other Languages

Most programming languages define the order in which statements are executed (in some languages that support a model of parallel execution, the order of some statement executions is indeterminate). Many of them have a more abstract view of program execution than C (which in many ways is more closely tied to host processor execution) and say little more than assignments update the value of a variable.

Coding Guidelines

Most order of execution coding problems occur because developers assumed an ordering that is not guaranteed by the standard; for instance, expressions containing multiple sequence points. The details of when this occurs is a developer education issue, not a coding guidelines issue (other coding guideline documents sometimes simply recommend against writing such expressions).

— At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred. 194

Commentary

This requirement ensures that if there is no more than one volatile access between consecutive sequence points, the order in which the accesses occur is the same as the order in which the sequence points are reached. Also this requirement deals with accesses, not values. Almost nothing can be said about the values of volatile objects.

Except for the sequence point that occurs at the semicolon punctuator, usually little can be said about the order in which sequence points occur.

sequence points 187

Common Implementations

Implementations usually treat expressions containing volatile objects with great caution. In some cases many optimizations are not performed for the full expression referencing such objects.

Coding Guidelines

Experience suggests that developers sometimes have their own beliefs on access ordering relationships that goes beyond the minimum requirements of the C Standard. These beliefs only become a potential problem when the same object is modified more than once between two sequence points.

sequence points 187

Example

In the following there is no guaranteed order of object access to `b` or `c` in the first statement. A comma operator provides the necessary sequence point in the second expression.

```
1  int a;
2  volatile int b, c;
3
4  void f(void)
5  {
6  int t;
7
8  a = b + c;
9  a = ((t = b), t + c);
10 }
```

while in:

```
1  int x,
2      y;
```

```
3   volatile int b, c;
4
5   void f(void)
6   {
7   x = b + c;
8   y = b + c;
9   }
```

the common subexpression b+c cannot be optimized in the assignment to y. Normally a translator would have the option of keeping the value of this calculation in a register, or loading it from x. However, because the operands have the volatile storage class, they must be accessed to obtain their values.

---

195 10) In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **main** will have ended in the former case, even where they would not have in the latter. footnote 10

C90

This footnote did not appear in the C90 Standard and was added by the response to DR #085.

Example

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int *pi;
5
6   void DR_085(void)
7   {
8   /*
9    * The lifetime of the object accessed by *pi
10   * has terminated.  Undefined behavior.
11   */
12  printf("Value is %d\n", *pi);
13  }
14
15  int main(void)
16  {
17  int i;
18
19  atexit(DR_085);
20  i = 42;
21  pi = &i;
22  return 0; /* Causes wording in DR #85 apply. */
23  }
```

---

196 11) The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. footnote 11

Commentary

The footnote is making the observation that modifying the values of these user-accessible status flags and control modes is to be considered a change of state of the execution environment. However, the response to DR #287 states that status flags are not objects (and modifying them between sequence points does not generate undefined behavior).

200 status flag floating-point
200 control mode floating-point
957 footnote DR287
941 object modified once between sequence points

> *When set, a status flag may contain additional system-dependent information, possibly inaccessible to some users. The operations of this standard may, as a side-effect, set some of the following flags: inexact result, underflow, overflow, divide by zero and invalid operation.*

IEC 60559

Inexact occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. This happens when the result of an operation overflows, or underflows, or is not exactly representable in the precision used. For instance, when `FLT_RADIX == 2`, the value of `1.0/10.0` is inexact, but is exact when `FLT_RADIX == 10` (in the past, computers sold into the business market, where division by powers of 10 is common, were often designed to use a base 10-radix for this reason). One use for this flag is to support integer arithmetic in a floating-point unit.

subnormal numbers 338

Underflow: the result of an arithmetic operation, needs to be represented as a subnormal number or between zero and the smallest subnormal number. Underflow can also cause an exception to be raised.[305]

Overflow: the result of an arithmetic operation, is greater than the largest finite floating-point number. For, divide-by-zero, as the name suggests, an attempt has been made to divide by zero (except for the special case where the numerator is also zero). It also occurs if an operation on a finite value yields an exact infinite result. Invalid operation— this might include subtracting infinity from infinity, or dividing infinity by infinity, or zero by zero.

IEC 60559 *The following modes shall be implemented:*

1. *rounding, to control the direction of rounding errors,*
2. *and in certain implementations, rounding precision, to shorten the precision of the result. [This is not required for chips that do: single = single OP single. It is required for chips that do everything in more precision than single, such as Intel x87].*
3. *The implementation may, at his option, implement the following modes: traps disabled/enabled, to handle exceptions.*

Warning: The IEC 559 Standard copied the preceding words from the IEEE-754 and IEEE-854 standards incorrectly, changing the meaning. Make sure you have a copy of the IEC 60559 document, not IEC 559.

C90

The dependence on this floating-point format is new in C99. But, it is still not required.

C++

The C++ Standard does not make these observations about IEC 60559.

Common Implementations

Most processors implement the status flags listed here. The function `_control87` was supported by many MS-DOS based (with continuing compatibility support under Windows) implementations hosted on Intel x86 processors. This function enabled various Intel x87 maths coprocessor (now integrated with the CPU) control flags to be read and set.

Most implementations assume that the developer will not change the control modes (such usage is rare and handling it can introduce a large performance penalty) and that the processor will be running in round-to-nearest and default precision. If the user changes either mode and then calls a library function, defined in the headers **`<math.h>`** or **`<stdio.h>`**, incorrect results could be returned. The better vendors may supply two versions of the library: one that assumes the floating-point environment is the default one, and a slower one that allows for the developer altering the environment.

Coding Guidelines

The setting of the status flags and control modes defined in IEC 60559 represents information about the past and future execution of a program. Floating-point operations are a technically complex subject and the extent to which developers or source code alter or test this information will depend on many factors. Apart from the general exhortation to developers to be careful and to make sure they know what they are doing, there is little of practical use that can be recommended.

Commentary

In a sense floating-point operations may have semantic similarities with accesses to objects declared with a **volatile** qualified type. 1476 type qualifier syntax

Integer operations may also set status flags. The difference between the two sets of flags is that the floating-point flags are sticky (i.e., once set, they stay set until they are explicitly reset). This property of IEC 60559 status flags is now a recognized state of the C abstract machine. 200 status flag floating-point

C++

The C++ Standard does not say anything about status flags in the context of side effects. However, if a C++ implementation supports IEC 60559 (i.e., `is_iec559` is true, 18.2.1.2p52) then floating-point operations will implicitly set the status flags (as required by that standard).

Coding Guidelines

Checking status flags after every floating-point operation usually incurs a significant performance penalty. The status flags were designed to be sticky to enable checks to be made after a series of floating-point operations. However, the more operations performed before a check is made, the less detailed information is available about where the potential problem occurred. The extent to which it is cost effective to use the information provided by the status flags is outside the scope of these coding guidelines.

Example

```
#include <fenv.h>

extern float f_glob;

void f(void)
{
fexcept_t status_info;
float f_loc = f_glob * f_glob;

fegetexceptflag(&status_info, FE_ALL_EXCEPT);
/*
 * Now check the information returned in status_info.
 */
}
```

---

198 modes affect result values of floating-point operations.

Commentary

The modes affect such things as the rounding of arithmetic operations. To some extent developers have some control over them through the use of the FENV_ACCESS pragma. 352 FLT_ROUNDS

C++

The C++ Standard does not say anything about floating-point modes in the context of side effects.

---

199 Implementations that support such floating-point state are required to regard changes to it as side effects—see annex F for details. side effect floating-point state

Commentary

In practice the status flags associated with floating-point state differ from those associated with other operations (e.g., the flags set as the result of integer operations, such as result is zero) in that they can affect the results of floating-point operations. A processor's ability to select how integer overflow is handled might, in theory, be considered a mode. However, in practice few processors provide such functionality.[284]

The FENV_ACCESS pragma provides a mechanism for developers to inform implementations that they are accessing the floating-point environment.

C++

The C++ Standard does not specify any such requirement.

Common Implementations

In a multiprocess environment any floating-point state flags have to be saved and restored on each context switch. This saving and restoring is usually handled by the operating system.

Processors rarely have any status flags that affect operations on integer types (although some processors have the ability to dynamically change their endianness, which is usually only done at boot time). For this reason, the C Standard limits its discussion to floating-point state.

endian 570

Coding Guidelines

An algorithm may depend on a particular floating-point state for its correct operation. Programs using such algorithms need to ensure that this state is set up as part of the initialization done prior to using the algorithm. They may also need to restore any previous state, if it was different from the one used for the algorithm.

The floating-point environment library **<fenv.h>** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases. 200

Commentary

status flag
floating-point exception
floating-point control mode
floating-point

The following except from "7.6 Floating-point environment <fenv.h>" of the library section defines the terms *floating-point environment*, *floating-point status flag*, *floating-point exception*, and *floating-point control mode*.

7.6 Floating-point environment **<fenv.h>**

> *A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information. A floating-point control mode is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.*[DR287a)]

The WG14 document N753 says:

WG14/N753

> *It is assumed that the integer and floating-point environments each consist of a control word and a status word. The status word contains bits (sticky flags) to indicate the state of past operations. The status word need not be a hardware register, it may be in memory and maintained by system software.*
>
> *The floating-point environment consists of:*

```
status
   sticky flags
      invalid
      div-by-zero
      overflow
      underflow
      inexact
   optional
      current operation being performed
      exception(s) of current operation
      exception(s) still pending
      operand values
      destination's precision
      rounded result
control
   rounding
   precision (optional)
   trap enable/disable (optional)
```

```
            invalid
            div-by-zero
            overflow
            underflow
            inexact
```

While the Rationale says:

Rationale

The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that can affect a program's results. Each such option's deviation from this specification should be well documented.

Dynamic vs. static modes

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings, which imposes intellectual as well as time and space overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time, which inhibits optimization.

C99 addresses these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of static modes with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEC 60559 standard requires dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, standard C has no facility, other than pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling `FENV_ACCESS` pragma would need such special treatment.

Translation

An implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program.

The language and library provide facilities to cause floating-point operations to be done at execution time when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.

C90

Support for the header **<fenv.h>** is new in C99.

C++

Support for the header **<fenv.h>** is new in C99, and there is no equivalent library header specified in the C++ Standard.

Other Languages

There is an ISO Technical Report[651] that deals with floating-point exception handling in Fortran.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced. 201

Commentary

A strictly conforming program has a unique sequence of output data for a given sequence of input data. The output from a conforming program is not guaranteed to be unique (an unspecified behavior may cause different implementations to generate different).

interactive device intent 203

The standard says very little about when output appears. Program termination causes all open files to be closed. Although closing a file causes any buffered data to be written to it, nothing is said about when this flushing needs to be completed. The only thing that can be said about program termination is that there will be no more output from that program (during that execution).

The library file-handling functions deal with the issue from a single executing program's perspective. How another program, running on a host capable of supporting more than one program executing at the same time, might view the contents of a file being written to by more than one conforming C program at the same time is not dealt with by the C Standard.

C++

1.9p11 — *At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.*

The C++ Standard is technically more accurate in recognizing that the output of a conforming program may vary, if it contains unspecified behavior.

Common Implementations

Many file systems' cache writes, to a file, into blocks. Once a block is full the data it contains is written to the file. Any cached data is also written to a file when it is closed. Many modern environments have multiple caches. Whether a buffer flushed from an individual program's I/O cache ever gets physically written, as a pattern of bits, on to an actual storage device might never be known. A temporary file may be opened, written to, closed and then removed; all associated data being held in one or more caches.

— The input and output dynamics of interactive devices shall take place as specified in 7.19.3. 202

Commentary

Clause 7.19.3 does not say what shall happen, only what is intended to happen.

Rationale The class of interactive devices is intended to include at least asynchronous terminals, or paired display screens and keyboards. An implementation may extend the definition to include other input and output devices, or even network inter-program connections, provided they obey the Standard's characterization of interactivity.

Common Implementations

Having the host processor, in a multiuser environment, deal with every key press from all users can consume a large amount of resources. Some mainframe environments require an escape key to be pressed to indicate to the host that input is being sent. It can be more efficient to have the local input device store the characters until a response is required from the host (e.g., at the end of a newly typed line of characters).

Most, nonmainframe, implementations can support a line-oriented conversation.

---

interactive device intent

The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input. 203

Commentary

Such buffering is very desirable when attempting to have a realtime conversation with a person or computer at the sending/receiving end of the input/output. Not all hosts provide the ability to perform anything other than buffered I/O, hence this specification is an intent rather than a requirement.

Line-buffered output is intended to appear every time a new-line character is written to a stream. The individual characters need not appear as they are written by the program, for instance if a string is output, followed several source statements later by the output of a number. The output might appear a line at a time.

The closing of a stream is another event that is likely to cause output to appear fairly promptly.

Common Implementations

Having the host dispatch output characters one at a time for writing to a file is very inefficient. In a multiuser environment there are performance gains to be had in passing complete lines to the output device.

There can be a significant performance penalty associated with continually opening and closing streams.

Coding Guidelines

I/O handling is an important issue for applications and vendors often put a lot of effort into this functionality. It is one area where extensions are often used and where following guidelines can help minimize dependencies on a particular implementation. The extent to which an application depends on output appearing in a timely manner is a design issue that is outside the scope of these coding guidelines.

95.1 extensions cost/benefit

Example

Once consequence of buffered output handling is that the same program may behave differently depending on how it is executed:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3
4  int main(void)
5  {
6  printf("Started\n");
7  if (fork() == 0)
8     printf("Child\n");
9  }
```

If, when executed, `stdout` is an interactive device, the preceding program will produce:

```
Started
Child
```

If, when executed, `stdout` is not an interactive device, there is a high probability that the program will produce:

```
Started
Started
Child
```

The reason is that the `fork` system call (it's in POSIX, but not C) duplicates all of the parents' data structures, including its internal I/O buffers (defined by the C header **<stdio.h>**) for the child process. If buffered I/O is taking place (likely when writing to a noninteractive device), the string generated by the first `printf` will be held in one of these buffers before being copied to the O/S output buffers. When the **<stdio.h>** buffers are copied, their contents, including any pending output, is also copied. When these buffers are finally flushed, two copies of the string **Started** appear on the output stream (see section 8.2 of Zlotnick[1515] for a detailed discussion and possible solutions).

---

204 What constitutes an interactive device is implementation-defined.

interactive device what constitutes

Commentary

Not only the type of device, but how it is used may affect whether it is treated as an interactive device. Serial devices (i.e., those that send and receive data as a stream of bytes) are often implemented as interactive devices. However, if access to a device is occurring as some background task because the program is executing in batch mode, the host may decide not to treat it as an interactive device.

Block devices (i.e., those that handle data in blocks, usually of 512 bytes or more) are not usually interactive devices. If more than one program is accessing data written to a block device, it may be necessary to ensure that all writes to that device occur when they are executed. But, support for such multiprogram behavior is outside the scope of the C Standard. One block device that is sometimes treated as an interactive device is a tape. Here the output is assembled into blocks for efficient writing and storage, but is written serially.

Other Languages

Most languages do not specify anything about low-level device details.

Common Implementations

The devices connected to `stdin`, `stdout`, and `stderr` on program startup are often interactive devices, if such are available.

Hard disks are block devices and are rarely interactive devices.

Coding Guidelines

Programs that require their output to appear in a timely fashion need to ensure that the devices they are using support such behavior. The C Standard provides intent to implementors, not guarantees to developers. Coding guidelines can say nothing, other than reminding developers that the intended behavior may vary between implementations.

---

semantics stringent correspondence

More stringent correspondences between abstract and actual semantics may be defined by each implementation. 205

Commentary

This is really a warning that implementations may have a low quality of implementation in this area. They might not perform any code optimizations, updating all objects at the point their values are modified; I/O could be performed on a character-by-character basis (input and output is defined in terms of `fgetc` and `fputc`).

Common Implementations

The commercial pressures on vendors is rarely to conform more strictly to standards (although a strong interest in conformance to standards is invariably claimed). Rather, it is to provide features that improve a program's ability to interact with the host environment. Many implementations provide several additional functions to ensure that characters are written to a device in a timely manner.

Coding Guidelines

More stringent correspondences would not change the behavior of a strictly conforming program. It may change the behavior of a conforming program to one of the set of possible behaviors that any implementation could produce.

A general principle of coding guidelines is to recommend usage that minimizes a program's exposure to the latitude available to an implementation in executing it. More stringent correspondences might be viewed as enabling programs to produce more reliable results. However, using a more stringent implementation only saves costs if the program has been written to that specification. Porting existing code to such an environment is simply that, another port.

---

abstract machine example

EXAMPLE 1 An implementation might define a one-to-one correspondence between abstract and actual 206

semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **volatile** would then be redundant.
Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the **signal** function would require explicit specification of **volatile** storage, as well as other implementation-defined restrictions.

Commentary

Use of **volatile** is usually treated as a very strong indicator that the designated object may change in unexpected ways. For it to be redundant the implementation would also have to specify the order of evaluations of an expression containing objects defined using this qualifier. If an optimizer were clever enough, it could even ignore these sequence points (which is leading edge optimizer technology). Commercially available optimizers tend to limit themselves to what they can analyze in a single translation unit.

Requiring that the program declare all objects accessed by an interrupt service routine with the **volatile** storage-class specifier is overly restrictive. Such routines can be invoked other than via a call to the `abort` or `raise` functions. The values of objects, as of the previous sequence point, can be relied on.

191 signal interrupt
abstract machine processing

Common Implementations

On average every fifth statement is a function call. This is very frustrating for writers of optimizers (long sequences of C code without any function calls provide more opportunities to generate high-quality machine code). The introduction of support for the **inline** function specifier, in C99, offers one way around this problem for time-critical code.

1522 function specifier
syntax

Cross function call optimization is made easier if all of the source of the called function is available (i.e., it is in the same translation unit as the caller), something that does not often occur in practice. Generation of the highest quality code requires that code generation be delayed until link-time, when all of the information about a program is available.[985, 1437]

The ability of an expression to raise a signal ruins the potential for optimization. For instance, in:

```
glob_flag = 1;

x=expr1_could_raise_signal;

glob_flag = 2;

x=expr2_could_raise_signal;

glob_flag = 3;
```

a function registered to handle the signal that may be raised in either expression might want to examine `glob_flag` to get some idea of which expression raised the signal.

Calls to the `raise` function can be detected and code generated to ensure that objects have the values required by the abstract machine.

---

207 EXAMPLE 2 In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the "integer promotions" require that the abstract machine promote the value of each variable to **int** size and then add the two **int**s and truncate the sum. Provided the addition of two **char**s can be done without overflow,

or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

Commentary

Silently wrapping to produce the correct result is the most commonly seen processor behavior. Because `c1` and `c2` have the same types in this example, there are no potential complications introduced by them having different signed types.

Common Implementations

Modern processor instructions invariably operate on the contents of registers. These having been zero filled or sign extended, if necessary, when the value was loaded from storage, irrespective of the size of the value loaded. The cast, for promoted values, is thus implicit in the load instruction.

Older processors (including the Intel x86 processor family) have the ability to load values into particular bytes of a register. It is possible for some instructions to operate on these subparts of a register. The original rationale of such a design is that instructions operating on smaller ranges of values could be made to execute faster than those operating on larger ranges. Technology has now advanced (the Intel x86 in particular) to the stage where there are no longer any performance advantages to such a design. The issue is now one of economics— wider processor buses incur higher costs.

Some implementations (e.g., Tasking[20]) support an option that causes operations on operands having character type to be performed using processor instructions that manipulate 8-bit values (i.e., the integer promotions are not performed).

Coding Guidelines

In expressions such as:

```
1  unsigned char c1, c2, c3;
2
3  void f(void)
4  {
5  if (c1 == c2 + c3)
6     c1=3;
7  }
```

there is a commonly encountered incorrect assumption made by developers that in this case the equality and arithmetic operations are all performed at character type, the result of the addition always being kept within range of the type **`unsigned char`**. Recommending that all developers' be trained on the default integer promotions and the usual arithmetic conversions is outside the scope of these coding guidelines. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults (perhaps resulting from poor training).

guidelines 0
not faults

EXAMPLE 3 Similarly, in the fragment 208

```
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
```

the multiplication may be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if **`d`** were replaced by the constant **`2.0`**, which has type **`double`**).

Commentary

Such ascertaining is very hard, in practice, to do for floating-point arithmetic. It is also very tempting (instructions that operate on single-precision values are sometimes much faster than those that operate on double-precision values; not on Intel x87 where operations on objects having type **`long double`** are fastest).

FLT_EVAL_METHOD 354

The precision in which floating-point operations take place is not just a matter of performance. A numerical

algorithm may depend on a particular level of accuracy. More-than-expected accuracy can sometimes be just as damaging to the final result as less-than-expected accuracy.

---

209 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **d1** and **d2** are required to have been converted to **float**.

Commentary

register spilling

Spilling is a technical code-generation term for what happens when the generator runs out of free working registers. It has to temporarily copy the contents of a register to a holding area in storage, freeing that register to be used to hold another value. Compiler writers hate register spills (not being able to generate code that only requires the available registers is seen as a weakness in the quality of their product).

There may not need to be an explicit store-and-load operation to round a value to the precision of the storage type. A processor instruction may be available to perform such as a conversion, which has the same effect. Even if they are converted back to the type **double** prior to the assignment, the sequence of cast operations `(double)(float)` is not a no-operation. The conversion to type **float** may result in a loss of precision in the value converted.

C90

This example is new in C99.

Common Implementations

The floating-point unit for the Intel x86 processor family[627] uses 80 bits internally to represent floating-point values. All floating-point operations are usually performed using this representation. A precision-control word allows this behavior to be changed during program execution for the floating add, subtract, multiple, divide, and square root instructions. However, this control word only affects the precision of the significand; the range of possible exponent values is not reduced (15 bits are always used). Because the range of exponent values is not reduced, it is possible to represent smaller values than would be possible in the single- or double-precision type. The only way to ensure that all additional bits used in the 80-bit representation are removed is to store the value, to storage, and reload it. The specification of the Java virtual machine requires that floating-point operations be carried out to the accuracy of the type, only. Performing the store/load operations needed to meet this requirement causes a factor of 10 performance penalty, although techniques to reduce this to a factor of 2 to 4 have been proposed.[665]

Some implementations (e.g., Microsoft Visual C++, `gcc`) provide an option that results in all reads of objects (having a real type) to be loaded from the object (all assignments also cause the object value to be updated). By not optimizing the generated machine code to reuse any value that happens to be in one of the floating-point registers, the consistency of expression evaluations is improved. If this option is not enabled, there is the possibility that a reference to an object will have greater precision in some cases because the value used was one already present in an 80-bit register, with potentially greater accuracy because it had been returned as the result of an arithmetic operation, and not loaded from storage.

A register spill has the potential for altering the value of a floating-point number. For instance, if registers hold values to greater precision than the representation of a subexpression's C type, the instructions used to perform the spill may chose to save the exact register contents to temporary storage or may round the result to the C type. On some processors the overhead of storing the exact register contents is much higher than using

the ordinary floating store instructions; some implementations have been known to chose to use the faster instructions, effectively rounding an intermediate result in those cases where a register spill had to be made.

Example

In the following:

```
#include <stdio.h>

double f3, f1, f2;

void f(void)
{
f3=f1+f2; /* The assignment. */

if (f3 == (f1+f2)) /* A simple comparison. */
   printf("As expected\n");
else
   printf("Did not expect to get here (but it can happen)\n");

if (f3 == (double)(f1+f2)) /* Another comparison. */
   printf("As expected\n");
else
   printf("This never appears\n");
}
```

the assignment will scrape off any extra bits that may be held by the processor performing the addition. In the simple comparison it would be tempting (and permitted by the standard) for an implementation, in terms of quality of generated machine code, to perform the addition of `f1` and `f2`, hold the result in a register, and then compare against `f3`. However, if the result of the addition is not adjusted to contain the same number of representation bits as were stored into `f3`, there is the possibility that any extra bits returned by the addition operation will cause the *another comparison* to fail.

EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.8). 210

```
double x, y, z;
/* ... */
x = (x * y) * z;  // not equivalent to x *= y * z;
z = (x - y) + y ; // not equivalent to z = x;
z = x + x * y;    // not equivalent to z = x * (1.0 + y);
y = x / 5.0;      // not equivalent to y = x * 0.2;
```

Commentary

The normal arithmetic identities that hold for unsigned integer types (and sometimes for signed integer types) rarely hold for floating-point types.

C90

This example is new in C99.

Other Languages

Fortran is known for its sophistication in handling floating-point calculations.

Common Implementations

The average application does not use floating-point types. Applications involving intensive floating-point calculations tend to be niche markets, and there are niche vendors who specialize in translating programs that perform lots of floating-point operations.

Coding Guidelines

The average translator is not very sophisticated with respect to optimizing expressions containing floating-point values. If possible, such optimizations need to be disabled unless purchasing a product from a vendor known to specialize in numerical computation.

---

211 EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment

EXAMPLE expression grouping

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum **(a + 32760)** is next added to **b**, and that result is then added to **5** which results in the value assigned to **a**. On a machine in which overflows produce an explicit trap and in which the range of values representable by an **int** is [-32768, +32767], the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for **a** and **b** were, respectively, -32754 and -15, the sum **a + b** would produce a trap while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for **a** and **b** might have been, respectively, 4 and -8 or -17 and 12. However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

Commentary

This example illustrates how a left-to-right parse of the input token stream associates the operands. The binding of operands to operators having the same precedence is specified by the language syntax. The brackets do not imply any ordering on the accessing of the objects a and b. An implementation may choose to load b into a register before evaluating (a+32760). Such behavior will not be noticeable unless both operands have a volatile-qualified type and their values change during execution.

943 expression grouping
943 operator precedence
1481 volatile qualified attempt modify

C90

The C90 Standard used the term *exception* rather than *trap*.

Common Implementations

Host processors that trap on overflow of signed integer operations are rare and many translators would freely rewrite the expression. Issues of instruction performance and implementation simplicity have won the day, in most cases.

---

212 EXAMPLE 7 The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of **p** can occur at any time between the previous sequence point and the next sequence point (the **;**), and the call to **getchar** can occur at any point prior to the need of its returned value.

Commentary

A C translator also has complete freedom, subject to sequence point requirements, to select the order in which the various parts of an expression are evaluated. Perhaps even reusing a result from a previously calculated subexpression.

Common Implementations

With optimizations switched off, many translators will generate code to evaluate the expression in either left-to-right, or right-to-left order.

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), the **signal** function (7.14), files (7.19.3). 213

5.2 Environmental considerations

Commentary

Rationale

The C89 Committee ultimately came to remarkable unanimity on the subject of character set requirements. There was strong sentiment that C should not be tied to Ascii, despite its heritage and despite the precedent of Ada being defined in terms of Ascii. Rather, an implementation is required to provide a unique character code for each of the printable graphics used by C, and for each of the control codes representable by an escape sequence. (No particular graphic representation for any character is prescribed; thus the common Japanese practice of using the glyph "¥" for the C character "\" is perfectly legitimate.) Translation and execution environments may have different character sets, but each must meet this requirement in its own way. The goal is to ensure that a conforming implementation can translate a C translator written in C.

For this reason, and for economy of description, source code is described as if it undergoes the same translation as text that is input by the standard library I/O routines: each line is terminated by some newline character regardless of its external representation.

5.2.1 Character sets

---

source character set
execution character set

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). 214

Commentary

This is a requirement on the implementation. This defines the terms *source character set* and *execution character set*.

A collating sequence is defined, but the particular values for the characters are not specified. These, and only these, characters are guaranteed to be supported by a conforming implementation.

The purpose for defining these two character sets is to separate out the environment in which the source is translated from the environment in which the translated output is executed. When these two environments are different, the translation process is commonly known as *cross compiling*. An implementation may add additional characters to either the source or execution character set. There is no requirement that any additional characters exist in either environment.

The characters used in the definition of the C language exist within both the source and execution character sets. It is intended that a C translator be able to successfully translate a C translator written in C.

Rationale

ISO (the International Organization for Standardization) uses three technical terms to describe character sets: repertoire, collating sequence, and codeset. The repertoire is the set of distinct printable characters. The term abstracts the notion of printable character from any particular representation; the glyphs R, R, R, *R*, **R**, R, and **R**, all represent the same element of the repertoire, "upper-case-R", which is distinct from "lower-case-r". Having decided on the repertoire to be used (C needs a repertoire of 91 characters plus whitespace), one can then pick a collating sequence which corresponds to the internal representation in a computer. The repertoire and collating sequence together form the codeset.

What is needed for C is to determine the necessary repertoire, ignore the collating sequence altogether (it is of no importance to the language), and then find ways of expressing the repertoire in a way that should give no problems with currently popular codesets.

C90

The C90 Standard did not explicitly define the terms *source character set* and *execution character set*.

C++

The C++ Standard does not contain a requirement to define a collating sequence on the character sets it specifies.

Other Languages

Many languages are designed for a hosted environment and do not need to make the distinction between source and execution character sets. Ada is explicitly defined in terms of the Ascii character set. After all, it was designed as a language for use by the US Department of Defense.

Common Implementations

On most implementations the two characters sets have the same representations.

Coding Guidelines

Developers whose native tongue is English tend to be unaware of the distinction between source and execution character sets. Most of these developers do most of their development in environments where they are identical.

---

215 Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*.

basic character set
extended characters

Commentary

This defines the terms *basic character set* and *extended characters*. It separates out the two components of the character set used by an implementation; the one which is always required to be provided and the extended set which is optional.

C90

This explicit subdivision of characters into sets is new in C99. The wording in the C90 Standard specified the minimum contents of the basic source and basic execution character sets. These terms are now defined exactly, with all other characters being called extended characters.

214 source character set

. . . ; any additional members beyond those required by this subclause are locale-specific.

C++

2.2p3 *The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.*

The C++ Standard more closely follows the C90 wording.

Other Languages

In the past most programming languages tended not say anything about supporting other character set members, although most implementations usually supported some additional characters. This situation is starting to change as the predominantly English-speaking standards world starts to recognize the importance of supporting programs written in the developer's native character set (the introduction of ISO 10646 also helped).

ISO 10646 28

Common Implementations

A large number of C translators originate in the USA, or target this market. This locale usually requires support for extended characters in the form of those members in the Ascii character set that are not in the basic source character set (the *$* and @ characters being the most obvious).

Vendors selling into non-English-speaking markets commonly add support for extended characters in the execution character set to support a native language. These implementations usually also support the occurrence of extended characters in comments. Support for extended characters outside of character constants, string literals, and comments has been much less common.

Coding Guidelines

Using extended characters to make applications comprehensible to their users is obviously essential. However, the handling of such characters is part of the application domain and outside the scope of these coding guidelines. The issues involved in programs written in one locale targeted at another locale is also largely outside the scope of these coding guidelines.

Using characters from the developer's native language can make an important contribution to program readability for those developers who share that native language. Some applications are now developed using people whose native languages differ from each other. The issue of using extended characters to improve source code readability is not always clear-cut. What is the best way to handle programs made up of translation units developed by developers from different locales; should all developers working on the same application use the same locale? To a large extent these questions involve predicting the future. Who will be doing the future development and maintenance of the software? It may not be possible to provide a reliable answer to this question. The issue of what characters to use in identifier names is discussed elsewhere.

identifier 792 syntax

Example

```
1  char dollar = '$';
```

---

extended character set

The combined set is also called the *extended character set*.

216

Commentary

A somewhat confusing use of terminology. A developer might be forgiven for thinking that this term applied to the set of extended characters only. For both the source character set and the execution character set, the following statement is true:

```
1  extended_character_set = basic_character_set + extended_characters;
```

Coding Guidelines

A coding guideline document needs to be very careful in its use of terminology when dealing with character set issues.

---

217 The values of the members of the execution character set are implementation-defined.

Commentary

This has already been stated for the members of the source character set. Although it might not specify their values, the standard does specify some of the properties of objects that hold them.

116 translation phase 1
222 basic character set fit in a byte

Common Implementations

Many implementations use the Ascii character set, with the EBCDIC character set appearing to be restricted to use on IBM mainframes and their derivatives. Most implementations use the same values for the corresponding members of both the source and execution character set.

Coding Guidelines

Developers tend to make several assumptions about the values of the execution character set:

- They are the same as the source character set.
- All the uppercase letters, all the lowercase letters, and all the digits are contiguous.
- They are less than 128.
- The actual values used by a translator (e.g., space being 32).

Only the assumption about the digits being contiguous is guaranteed to be true.

223 digit characters contiguous

A program may contain implicit dependencies on the representation of members of the execution character set because developers are not aware they are making assumptions about something that is not fixed. Designing programs to accommodate the properties of different character sets is not a trivial matter that can be covered in a few guideline recommendations.

Example

```
#if 'a' == 99 /* Not the execution character set. */
#endif

int f(void)
{
return 'b' == 88;
}
```

---

218 In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash \ followed by one or more characters.

execution character set represented by

Commentary

This describes the two methods specified by the standard for representing members of the execution character set in character constants and string literals. They are one route through which characters appearing in the source code can appear in the output produced by a program. Comments are removed in translation phase 3. Identifier spellings are represented by objects in the program image, and are not directly involved in execution-time behavior.

126 comment replaced by space

Escape sequences are a method of representing execution characters in the source, which may not be representable in the source character set. They make it possible to explicitly specify a particular numeric value, which is known to represent a given character in the execution character set (as defined by the implementation).

Other Languages

The convention of mapping source characters to their corresponding execution characters is common to the majority of programming languages. Some of the more recently designed, or updated, programming languages also support some form of escape sequence mechanism.

Common Implementations

The POSIX locale mechanism defines a representation for characters based on their names. For instance, *LETTER-A* is used to denote the character *A*. This approach removes the need for representing characters on keyboards and displays. The main use, to date, for this character specification methodology has been within POSIX locale specifications, where the meaning of a sequence of one or more characters might otherwise be uncertain.

Coding Guidelines

String literals are not always used to simply represent character sequences. A developer may choose to embed other, numeric, information within a string literal. Relying on characters to have the desired value would create a dependence on a particular character set being used and create literals that were harder to interpret (use of escape sequences makes it explicit that a numeric value is required). The contents of string literals therefore need to be interpreted in the context in which they are used.

The value of an escape sequence may, or may not, be the same as that of a member of the basic character set. The extent to which the value of an escape sequence does, or does not, represent a member of the basic character set is one of intent on the part of the developer. This issue is discussed elsewhere.

escape sequence 866 syntax

Example

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  printf("\110ello World\n"); /* ASCII \110 == H */
6  printf("Be \100 one\n");    /* @ symbol not on an ASCII keyboard. */
7  }
```

---

null character

A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set; 219

Commentary

This defines the term *null character* (an equivalent one for a null wide character is given in the library section). The null character is used to terminate string literals.

string literal 902 zero appended

The C committee once received a request from a communications-related standards committee asking that this requirement be removed from the C Standard. The sending of null bytes was causing problems on some communications links. The C committee pointed out that C's usage was a long-established practice and that they had no plans to change it.

Other Languages

Some form of null character occurs in any language that uses a terminating character (rather than a length count) to represent strings. SQL has a null value. It is used to indicate *value unknown*, or *value not present* (no two NULL values can ever compare equal).

Coding Guidelines

There is a common beginner's mistake that is sometimes not diagnosed because an implementation has defined the `NULL` macro to be `0`, rather than `(void *)0`. If C++ compatible headers are being used, the problem is not helped by that language's explicit requirement that the null pointer be represented by `0`.

220 it is used to terminate a character string.

character string terminate

Commentary

A string literal may contain more than one null character, or none at all. In the former case the literal will contain more than one string (according to the definition of that term given in the library section) — for instance, `"abc\000xyz"` and `char s[3] = "abc"`. Each null character terminates a string even though more than one of them may appear in a string literal.

C++

After any necessary concatenation, in translation phase 7 (2.1), '\0' is appended to every string literal so that programs that scan a string can find its end. 2.13.4p4

In practice the C usage is the same as that specified by C++.

Other Languages

Not all languages specify a particular representation for strings. Some implementations of these languages use a numeric count, usually stored before the first character of the string, representation to specify the length of the string. Many Pascal implementations use this approach.

In some languages the representation of strings is completely hidden from the developer. Perl and Snobol have sophisticated built-in support for string creation and manipulation. Their implementations can use whatever representation techniques they choose. Programs attempting to access the representation, for this kind of language, are failing to operate in the algorithmic domain that the language designers intended.

Common Implementations

The MrC[263] compiler from Apple Computer provides the escape sequence \p so that developers can specify that a string literal is a *Pascal string*. It has to occur as the first character and causes the implementation to maintain a byte count, rather than a null terminator.

Coding Guidelines

Null characters are different from other escape sequences in a string literal in that they have the special status of acting as a terminator (e.g., the library string searching routines terminate when a null character is encountered, leaving any subsequent characters in the literal unchecked). Any surprising behavior occurring because of this usage is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

0 guidelines not faults

221 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet

basic source character set
basic execution character set

```
A  B  C  D  E  F  G  H  I  J  K  L  M
N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

the 26 *lowercase letters* of the Latin alphabet

```
a  b  c  d  e  f  g  h  i  j  k  l  m
n  o  p  q  r  s  t  u  v  w  x  y  z
```

the 10 decimal *digits*

```
0  1  2  3  4  5  6  7  8  9
```

the following 29 graphic characters

```
!  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed.

Commentary

The original C Standard's work aimed to codify existing practice, and the K&R definition used the preceding collection of characters. The character values occupied by the #, [,], {, }, & and | characters in the Ascii character set are sometimes used to represent different characters in some Scandinavian character sets. Whether an awareness of this issue would have made any difference to the characters chosen can be debated (as could the extent to which a problem experienced by a minority of developers should have any noticeable impact on the majority of developers). It certainly made no difference to the design of Java, which occurred well after this issue had become well-known.

The characters vertical tab, form feed, carriage return, and new-line are sometimes referred to as line break characters. This term describes the most commonly seen visual effect of their appearance in a text file, not how a translator is required to interpret them.

C90

The C90 Standard referred to these characters as the *English alphabet*.

C++

2.2p1 *The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphics characters:*

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

basic execution character set 226
control characters

The C++ Standard includes new-line in the basic source character set (C only includes it in the basic execution character set).

The C++ Standard does not separate out the uppercase, lowercase, and decimal digits from the graphical characters, so technically they are not defined for the basic source character set (the library functions such as `toupper` effectively define these terms for the execution character set).

Other Languages

Not all languages require as many characters to be supported as C does. The APL language contains so many characters especially designed for the language that they have their own standard – ISO 2575 *Registered Character Set 68— APL* – to describe them.

Common Implementations

The basic execution character set does not include three printable characters that appear in the first 127 positions of the ISO 10646 standard (and also in Ascii)— *$* (dollar), @ (commercial at), and ‘ (grave accent)

Coding Guidelines

A high priority might be given to supporting the *$* and @ characters in a money-oriented, wired world. But what about other characters that are not in the basic character set? When these characters are intended to appear in the source character set, the issue is one of displaying and potentially inputting them to a source file. Experience shows that developers usually have access to computers capable of displaying and inputting characters from the Ascii character set. This is an issue that is considered to be outside the scope of these coding guidelines. When these characters are intended to appear in the execution character set, it becomes an applications issue that is outside the scope of these coding guidelines.

Horizontal tab is a single white-space character. However, when viewing source code containing such a character, many display devices appear to replace it with more than one white-space character. There is no agreed-on spacing for the horizontal tab character and its use can cause the appearance of the source code to vary between display devices. The standard contains an alternative method of representing horizontal tab in string literals and character constants.

horizontal tab escape sequence 263

Most source code is displayed using a fixed-width font. Research[108] has shown that people read text faster (a 6% time difference, most of which can be attributed to the greater amount of information that a variable-width font allows to appear within the readers visual field) when it is displayed in a variable-width font than a fixed-width font. Comparing text has also been found to be quicker, but not searching for specific words. Use of a variable width font would also enable more characters to be displayed on a line, reducing the need to split statements across more than one line.

770 visual field

Support for use of variable-width fonts is not always available to developers. The issue of source, written using a variable-width font, having to be read in an environment where only a fixed-width font is available also needs to be considered (long lines may not be displayed as intended).[221.1]

Table 221.1: Occurrence of characters as a percentage of all characters and as a percentage of all noncomment characters (i.e., outside of comments). Based on the visible form of the `.c` files. For a comparison of letter usage in English language and identifiers see Figure 792.16.

Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment
0	0.000	0.000	**sp**	15.083	13.927	@	0.009	0.002	‘	0.004	0.002
1	0.000	0.000	**!**	0.102	0.127	**A**	0.592	0.642	**a**	3.132	2.830
2	0.000	0.000	**"**	0.376	0.471	**B**	0.258	0.287	**b**	0.846	0.812
3	0.000	0.000	**#**	0.175	0.219	**C**	0.607	0.663	**c**	2.168	2.178
4	0.000	0.000	**$**	0.005	0.003	**D**	0.461	0.523	**d**	2.184	2.176
5	0.000	0.000	**%**	0.105	0.135	**E**	0.869	1.012	**e**	5.642	4.981
6	0.000	0.000	**&**	0.237	0.311	**F**	0.333	0.355	**f**	1.666	1.725
7	0.000	0.000	**’**	0.101	0.080	**G**	0.243	0.263	**g**	0.923	0.906
8	0.000	0.000	**(**	1.372	1.751	**H**	0.146	0.155	**h**	1.145	0.777
\t	3.350	4.116	**)**	1.373	1.751	**I**	0.619	0.643	**i**	3.639	3.469
\n	3.630	4.229	*****	1.769	0.769	**J**	0.024	0.026	**j**	0.074	0.077
11	0.000	0.000	**+**	0.182	0.233	**K**	0.098	0.116	**k**	0.464	0.481
12	0.003	0.004	**,**	1.565	1.914	**L**	0.528	0.609	**l**	2.033	1.915
\r	0.001	0.001	**-**	1.176	0.831	**M**	0.333	0.366	**m**	1.245	1.229
14	0.000	0.000	**.**	0.512	0.387	**N**	0.557	0.610	**n**	3.225	2.989
15	0.000	0.000	/	0.718	0.519	**O**	0.467	0.517	**o**	2.784	2.328
16	0.000	0.000	**0**	1.465	1.694	**P**	0.460	0.508	**p**	1.505	1.551
17	0.000	0.000	**1**	0.502	0.551	**Q**	0.033	0.037	**q**	0.121	0.135
18	0.000	0.000	**2**	0.352	0.408	**R**	0.652	0.729	**r**	3.405	3.254
19	0.000	0.000	**3**	0.227	0.262	**S**	0.691	0.758	**s**	3.166	2.961
20	0.000	0.000	**4**	0.177	0.203	**T**	0.686	0.740	**t**	4.566	4.200
21	0.000	0.000	**5**	0.149	0.171	**U**	0.315	0.349	**u**	1.575	1.510
22	0.000	0.000	**6**	0.176	0.209	**V**	0.128	0.149	**v**	0.662	0.682
23	0.000	0.000	**7**	0.131	0.144	**W**	0.131	0.135	**w**	0.494	0.385
24	0.000	0.000	**8**	0.184	0.207	**X**	0.213	0.254	**x**	0.870	1.002
25	0.000	0.000	**9**	0.128	0.122	**Y**	0.091	0.094	**y**	0.515	0.435
26	0.000	0.000	**:**	0.192	0.176	**Z**	0.027	0.033	**z**	0.125	0.135
27	0.000	0.000	**;**	1.276	1.670	**[**	0.163	0.210	**{**	0.303	0.401
28	0.000	0.000	**<**	0.118	0.147	\	0.097	0.126	**|**	0.098	0.124
29	0.000	0.000	**=**	1.039	1.042	**]**	0.163	0.210	**}**	0.303	0.401
30	0.000	0.000	**>**	0.587	0.762	**^**	0.003	0.002	~	0.009	0.012
31	0.000	0.000	**?**	0.022	0.019	**_**	2.550	3.238	**127**	0.000	0.000

[221.1] Note that source code displayed in this book uses a fixed-width font. This usage is intended to act as a visual aid in distinguishing code from text and not as an endorsement of fixed-width fonts.

Table 221.2: Relative frequency (most common to least common, with parenthesis used to bracket extremely rare letters) of letter usage in various human languages (the English ranking is based on the British National Corpus). Based on Kelk.[718]

Language	Letters
English	etaoinsrhldcumfpgwybvkxjqz
French	esaitnrulodcmpévqfbghjàxèyêzâçîùôûïkëw
Norwegian	erntsilakodgmvfupbhøjyåæcwzx(q)
Swedish	eantrsildomkgväfhupåöbcyjxwzéq
Icelandic	anriestuðlgmkfhvoáþídjóbyæúöpéýcxwzq
Hungarian	eatlnskomzrigáéydbvhjőfupöócűíúüxw(q)

basic character set fit in a byte

The representation of each member of the source and execution basic character sets shall fit in a byte. 222

Commentary

character 59 single-byte character constant 883 type basic character set 478 positive if stored in char object

This is a requirement on the implementation. The definition of character already specifies that it fits in a byte. However, a character constant has type **int**; which could be thought to imply that the value representation of characters need not fit in a byte. This wording clarifies the situation. The representation of members of the basic execution character set is also required to be a nonnegative value.

C++

1.7p1 *A byte is at least large enough to contain any member of the basic execution character set and . . .*

This requirement reverses the dependency given in the C Standard, but the effect is the same.

Common Implementations

On hosts where characters have a width 16 or 32 bits, that choice has usually been made because of addressability issues (pointers only being able to point at storage on 16- or 32-bit address boundaries). It is not usually necessary to increase the size of a byte because of representational issues to do with the character set.

CHAR_BIT 307 macro

In the EBCDIC character set, the value of `'a'` is 129 (in Ascii it is 97). If the implementation-defined value of `CHAR_BIT` is 8, then this character, and some others, will not be representable in the type **signed char** (in most implementations the representation actually used is the negative value whose least significant eight bits are the same as those of the corresponding bits in the positive value, in the character set). In such implementations the type **char** will need to have the same representation as the type **unsigned char**.

The ICL 1900 series used a 6-bit byte. Implementing this requirement on such a host would not have been possible.

representation information using 569.1

Coding Guidelines

A general principle of coding guidelines is to recommend against the use of representation information. In this case the standard is guaranteeing that a character will fit within a given amount of storage. Relying on this requirement might almost be regarded as essential in some cases.

Example

```
1  void f(void)
2  {
3  char C_1 = 'W';        /* Guaranteed to fit in a char. */
4  char C_2 = '$';        /* Not guaranteed to fit in a char. */
5  signed char C_3 = 'W'; /* Not guaranteed to fit in a signed char. */
6  }
```

223 In both the source and execution basic character sets, the value of each character after **0** in the above list of decimal digits shall be one greater than the value of the previous.

digit characters contiguous

Commentary

This is a requirement on the implementation. The Committee realized that a large number of existing programs depended on this statement being true. It is certainly true for the two major character sets used in the English-speaking world, Ascii, EBCDIC, and all of the human language digit encodings specified in Unicode, see Table 797.1. The Committee thus saw fit to bless this usage.

Not only is it possible to perform relational comparisons on the digit characters (e.g, `'0'<'1'` is always true) but arithmetic operations can also be performed (e.g., `'0'+1 == '1'`). A similar statement for the alphabetic characters cannot be made because it would not be true for at least one character set in common use (e.g., EBCDIC).

C++

The above wording has been proposed as the response to C++ DR #173.

Other Languages

Most languages that have not recently had their specifications updated do not specify any representational properties for the values of their execution character sets. Java specifies the use of the Unicode character set (newer versions of the language specify newer versions of the Unicode Standard; all of which are the same as Ascii for their first 128 values), so this statement also holds true. Ada specifies the subset of ISO 10646 known as the Basic Multilingual Plane (the original language standard specified ISO 646).

28 ISO 10646

Coding Guidelines

This requirement on an implementation provides a guarantee of representation information that developers can make use of (e.g., in relational comparisons, see Table 866.3). The following are suggested wordings for deviations from the guideline recommendation dealing with making use of representation information.

569.1 representation information using

Dev 569.1 An integer character constant denoting a digit character may appear in the visible source as the operand of an *additive-operator*.

Example

```
1  #include <stdio.h>
2
3  extern char c_glob = '4';
4
5  int main(void)
6  {
7  if ('0' + 3 == '3')
8     printf("Sentence 221 is TRUE\n");
9
10 if (c_glob < '5')
11    printf("Sentence 221 may be TRUE\n");
12 if (c_glob < 53) /* '5' == 53 in ASCII */
13    printf("Sentence 221 does not apply\n");
14 }
```

224 In source files, there shall be some way of indicating the end of each line of text;

end-of-line representation

Commentary

This is a requirement on the implementation.

The C library makes a distinction between text and binary files. However, there is no requirement that source files exist in either of these forms. The worst-case scenario: In a host environment that did not have a native method of delimiting lines, an implementation would have to provide/define its own convention and supply tools for editing such files. Some integrated development environments do define their own conventions for storing source files and other associated information.

C++

The C++ Standard does not specify this level of detail (although it does refer to end-of-line indicators, 2.1p1n1).

Common Implementations

Unicode Technical Report #13: "Unicode newline guidelines" discusses the issues associated with representing new-lines in files. The ISO 6429 standard also defines NEL (NExt Line, hexadecimal 0x85) as an end-of-line indicator. The Microsoft Windows convention is to indicate this end-of-line with a carriage return/line feed pair, \r\n (a convention that goes back through CP/M to DEC RT-11); the Unix convention is to use a single line feed character \n; the MacIntosh convention is to use the carriage return character, \r.

Some mainframes implement a form of text files that mimic punched cards by having fixed-length lines. Each line contains the same number of characters, often 80. The space after the last user-written character is sometimes padded with spaces, other times it is padded with null characters.

---

this International Standard treats such an end-of-line indicator as if it were a single new-line character. 225

Commentary

translation phase 1 (116)

The standard is not interested in the details of the byte representation of end-of-line on storage media. It makes use of the concept of end-of-line and uses the conceptual simplification of treating it as if it were a single character.

C++

2.1p1n1 . . . *(introducing new-line characters for end-of-line indicators)* . . .

---

basic execution character set control characters

In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. 226

Commentary

This is a requirement on the implementation.

basic execution character set (221), translation phase 1 (116), escape sequence syntax (866)

These characters form part of the set of 96 execution character set members (counting the null character) defined by the standard, plus new line which is introduced in translation phase 1. However, these characters are not in the basic source character set, and are represented in it using escape sequences.

Other Languages

Few other languages include the concept of control characters, although many implementations provide semantics for them in source code (they are usually mapped exactly from the source to the execution character set). Java defines the same control characters as C and gives them their equivalent Ascii values. However, it does not define any semantics for these characters.

Common Implementations

ECMA-48 Control Functions for Coded Character Sets, Fifth Edition (available free from their Web site, http://www.ecma-international.ch) was fast-tracked as the third edition of ISO/IEC 6429. This standard defines significantly more control functions than those specified in the C Standard.

227 If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined.

Commentary

The standard does not prohibit such characters from occurring in a source file outright. The Committee was aware of implementations that used such characters to extend the language. For instance, the use of the @ character in an object definition to specify its address in storage.

The list of exceptions is extensive. The only usage remaining, for such characters, is as a punctuator. *Any other character* has to be accepted as a preprocessing token. It may subsequently, for instance, be stringized. It is the attempt to convert this preprocessing token into a token where the undefined behavior occurs.

1950 # operator
137 preprocessing token converted to token

C90

Support for additional characters in identifiers is new in C99.

C++

Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character. 2.1p1

The C++ Standard specifies the behavior and a translator is required to handle source code containing such a character. A C translator is permitted to issue a diagnostic and fail to translate the source code.

Other Languages

Most languages regard the appearance of an unknown character in the source as some form of error. Like C, most language implementations support additional characters in string literals and comments.

Common Implementations

Most implementations generate a diagnostic, either when the preprocessing token containing one of these characters is converted to a token, or as a result of the very likely subsequent syntax violation. Some implementations[717] define the @ character to be a token, its usual use being to provide the syntax for specifying the address at which an object is to be placed in storage. It is generally followed by an integer constant expression.

Coding Guidelines

An occurrence of a character outside of the basic source character set, in one of these contexts, is most likely to be a typing mistake and is very likely to be diagnosed by the translator. The other possibility is that such characters were intended to be used because use is being made of an extension. This issue is discussed elsewhere.

95.1 extensions cost/benefit

Example

```
1  static int glob @ 0x100; /* Put glob at location 0x100. */
```

228 A *letter* is an uppercase letter or a lowercase letter as defined above; letter

Commentary

This defines the term *letter*.

There is a third kind of case that characters can have, *titlecase* (a term sometimes applied to words where the first letter is in uppercase, or titlecase, and the other letters are in lowercase). In most instances titlecase is the same as uppercase, but there are a few characters where this is not true; for instance, the titlecase of the Unicode character U01C9, *lj*, is U01C8, *Lj*, and its uppercase is U01C7, *LJ*.

C90

This definition is new in C99.

---

in this International Standard the term does not include other characters that are letters in other alphabets. 229

Commentary

All implementations are required to support the basic source character set to which this terminology applies. Annex D lists those universal character names that can appear in identifiers. However, they are not referred to as letters (although they may well be regarded as such in their native language).

orthography 792

The term *letter* assumes that the orthography (writing system) of a language has an alphabet. Some orthographies, for instance Japanese, don't have an alphabet as such (let alone the concept of upper- and lowercase letters). Even when the orthography of a language does include characters that are considered to be matching upper and lowercase letters by speakers of that language (e.g., æ and Æ, å and Å), the C Standard does not define these characters to be *letter*s.

C++

The definition used in the C++ Standard, 17.3.2.1.3 (the footnote applies to C90 only), implies this is also true in C++.

Coding Guidelines

The term *letter* has a common usage meaning in a number of different languages. Developers do not often use this term in its C Standard sense. Perhaps the safest approach for coding guideline documents to take is to avoid use of this term completely.

The universal character name construct provides a way to name other characters. 230

Commentary

ISO 10646 28

In theory all characters on planet Earth and beyond. In practice, those defined in ISO 10646.

C90

Support for universal character names is new in C99.

Other Languages

Other language standards are slowly moving to support ISO 10646. Java supports a similar concept.

Common Implementations

Support for these characters is relatively new. It will take time before similarities between implementations become apparent.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1). 231

5.2.1.1 Trigraph sequences

trigraph sequences replaced by

~~All occurrences in a source file~~ Before any other processing takes place, each occurrence of one of the following sequences of three characters (called *trigraph sequences*[12)]) are replaced with the corresponding single character. 232

Commentary

Trigraphs were an invention of the C committee. They are a method of supporting the input (into source files, not executing programs) and the printing of some C source characters in countries whose alphabets, and keyboards, do not include them in their national character set. Digraphs, discussed elsewhere, are another sequence of characters that are replaced by a corresponding single character.

digraphs 916

string literal 895 syntax

The \? escape sequence was introduced to allow sequences of ?s to occur within string literals.

The wording was changed by the response to DR #309.

Other Languages

Until recently many computer languages did not attempt to be as worldly as C, requiring what might be called an *Ascii keyboard*. Pascal specifies what it calls lexical alternatives for some lexical tokens. The character sequences making up these lexical alternatives are only recognized in a context where they can form a single, complete token.

Common Implementations

On the Apple MacIntosh host, the notation `'????'` is used to denote the unknown file type. Translators in this environment often disable trigraphs by default to prevent unintended replacements from occurring.

---

trigraph sequences mappings

233

```
??= #   ??) ]   ??! |
??( [   ??' ^   ??< }
??/ \   ??< {   ??- ~
```

Commentary

The above sequences were chosen to minimize the likelihood of breaking any existing, conforming, C source code.

Other Languages

Many languages use a small subset, or none, of these problematic source characters, reducing the potential severity of the problem. The Pascal standard specifies `(.` and `.)` as alternative lexical representations of `[` and `]` respectively.

Common Implementations

Recognizing trigraph sequences entails a check against every character read in by the translator. Performance profiling of translators has shown that a large percentage of time is spent in the lexer. A study by Waite[1436] found 41% of total translation time was spent in a handcrafted lexer (with little code optimization performed by the translator). An automatically produced lexer, the `lex` tool was used, consumed 3 to 5 as much time.

One vendor, Borland, who used to take pride, and was known, for the speed at which their translators operated, did not include trigraph processing in the main translator program. A standalone utility was provided to perform trigraph processing. Those few programs that used trigraphs needed to be processed by this utility, generating a temporary file that was processed by the main translator program. While using this pre-preprocessor was a large overhead for programs that used trigraphs, performance was not degraded for source code that did not contain them.

Usage

There are insufficient trigraphs in the visible form of the `.c` files to enable any meaningful analysis of the usage of different trigraphs to be made.

---

234 No other trigraph sequences exist.

trigraph sequences no other

Commentary

The set of characters for which trigraphs were created to provide an alternative spelling are known, and unlikely to be extended.

Coding Guidelines

Although no other trigraph sequences exist, sequences of two adjacent questions marks in string literals may lead to confusion. Developers may be unsure about whether they represent a trigraph or not. Using the escape sequence `\?` on at least one of these questions marks can help clarify the intent.

Example

```
1  char *unknown_trigraph = "??++";
2  char *cannot_be_trigraph = "?\?--";
```

Usage

The visible form of the `.c` files contained 593 (`.h` 10) instances of two question marks (i.e., ??) in string literals that were not followed by a character that would have created a trigraph sequence.

Each `?` that does not begin one of the trigraphs listed above is not changed. 235

Commentary

Two ?s followed by any other character than those listed above is not a trigraph.

Common Implementations

No implementation is known to define any other sequence of ?s to be replaced by other characters.

Coding Guidelines

No other trigraph sequences are defined by the standard, have been notified for future addition to the standard, or used in known implementations. Placing restrictions on other uses of other sequences of ?s provides no benefit.

EXAMPLE 1 236

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

Commentary

This example was added by the response to DR #310 and is intended to show a common trigraph usage.

EXAMPLE 2 The following source line 237

```
printf("Eh???/n");
```

becomes (after replacement of the trigraph sequence `??/`)

```
printf("Eh?\n");
```

Commentary

This illustrates the sometimes surprising consequences of trigraph processing.

5.2.1.2 Multibyte characters

---

multibyte character source contain

The source character set may contain multibyte characters, used to represent members of the extended character set. 238

Commentary

multibyte 60 character transla- 116 tion phase 1 UCN 115 models of

The mapping from physical source file multibyte characters to the source character set occurs in translation phase 1. Whether multibyte characters are mapped to UCNs, single characters (if possible), or remain as multibyte characters depends on the model used by the implementation.

C++

The representations used for multibyte characters, in source code, invariably involve at least one character that is not in the basic source character set:

2.1p1 *Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.*

The C++ Standard does not discuss the issue of a translator having to process multibyte characters during translation. However, implementations may choose to replace such characters with a corresponding universal-character-name.

Other Languages

Most programming languages do not contain the concept of multibyte characters.

Common Implementations

Support for multibyte characters in identifiers, using a shift state encoding, is sometimes seen as an extension. Support for multibyte characters in this context using UCNs is new in C99. The most common implementations have been created to support the various Japanese character sets.

815 universal character name syntax

Coding Guidelines

The standard does not define how multibyte characters are to be represented. Any program that contains them is dependent on a particular implementation to do the right thing. Converting programs that existed before support for universal character names became available may not be economically viable.

Some coding guideline documents recommend against the use of characters that are not specified in the C Standard. Simply prohibiting multibyte characters because they rely on implementation-defined behavior ignores the cost/benefit issues applicable to the developers who need to read the source. These are complex issues for which your author has insufficient experience with which to frame any applicable guideline recommendations.

---

239 The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set.

Commentary

Multibyte characters could be read from a file during program execution, or even created by assigning byte values to contiguous array elements. These multibyte sequences could then be interpreted by various library functions as representing certain (wide) characters.

The execution character set need not be fixed at translation time. A program's locale can be changed at execution time (by a call to the `setlocale` function). Such a change of locale can alter how multibyte characters are interpreted by a library function.

C++

There is no explicit statement about such behavior being permitted in the C++ Standard. The C header **<wchar.h>** (specified in Amendment 1 to C90) is included by reference and so the support it defines for multibyte characters needs to be provided by C++ implementations.

Other Languages

Most languages do not include library functions for handling multibyte characters.

Coding Guidelines

Use of multibyte characters during program execution is an applications issue that is outside the scope of these coding guidelines.

---

240 For both character sets, the following shall hold:

Commentary

This is a set of requirements that applies to an implementation. It is the minimum set of guaranteed requirements that a program can rely on.

Coding Guidelines

The set of requirements listed in the following C-sentences is fairly general. Dealing with implementations that do not meet the requirements listed in these sentences is outside the scope of these coding guidelines.

---

241 — The basic character set shall be present and each character shall be encoded as a single byte.

Commentary

This is a requirement on the implementation. It prevents an implementation from being purely multibyte-based. The members of the basic character set are guaranteed to always be available and fit in a byte.

basic character set 222
fit in a byte

Common Implementations

An implementation that includes support for an extended character set might choose to define CHAR_BIT to be 16 (most of the commonly used characters in ISO 10646 are representable in 16 bits, each in UTF-16; at least those likely to be encountered outside of academic research and the traditional Chinese written on Hong Kong). Alternatively, an implementation may use an encoding where the members of the basic character set are representable in a byte, but some members of the extended character set require more than one byte for their encoding. One such representation is UTF-8.

extended character set 216
CHAR_BIT 307
macro
ISO 10646 28
UTF-16 28
UTF-8 28

— The presence, meaning, and representation of any additional members is locale-specific. 242

Commentary

On program startup the execution locale is the "C" locale. During execution it can be set under program control. The standard is silent on what the translation time locale might be.

Common Implementations

The full Ascii character set is used by a large number of implementations.

Coding Guidelines

It often comes as a surprise to developers to learn what characters the C Standard does not require to be provided by an implementation. Source code readability could be affected if any of these additional members appear within comments and cannot be meaningfully displayed. Balancing the benefits of using additional members against the likelihood of not being able to display them is a management issue.

The use of any additional members during the execution of a program will be driven by the user requirements of the application. This issue is outside the scope of these coding guidelines.

---

multibyte character
state-dependent encoding
shift state

— A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. 243

Commentary

State-dependent encodings are essentially finite state machines. When a state encoding, or any multibyte encoding, is being used the number of characters in a string literal is not the same as the number of bytes encountered before the null character. There is no requirement that the sequence of shift states and characters representing an extended character be unique.

extended characters 215

combining characters

There are situations where the visual appearance of two or more characters is considered to be a single character. For instance, (using ISO 10646 as the example encoding), the two characters *LATIN SMALL LETTER O* (U+006F) followed by *COMBINING CIRCUMFLEX ACCENT* (U+0302) represent the grapheme cluster (the ISO 10646 term[329] for what might be considered a *user character*) **ô** not the two characters **o ^**. Some languages use grapheme clusters that require more than one combining character, for instance ô̲. Unicode (not ISO 10646) defines a canonical accent ordering to handle sequences of these combining characters. The so-called *combining characters* are defined to combine with the character that comes immediately before them in the character stream. For backwards compatibility with other character encodings, and ease of conversion, the ISO 10646 Standard provides explicit codes for some accent characters; for instance, *LATIN SMALL LETTER O WITH CIRCUMFLEX* (U+00F4) also denotes **ô**.

A character that is capable of standing alone, the **o** above, is known as a *base* character. A character that modifies a base character, the **ô** above, is known as a *combining* character (the visible form of some combining characters are called *diacritic* characters). Most character encodings do not contain any combining characters, and those that do contain them rarely specify whether they should occur before or after the modified base

character. Claims that a particular standard require the combining character to occur before the base character it modifies may be based on a misunderstanding. For instance, ISO/IEC 6937 specifies a single-byte encoding for base characters and a double-byte encoding for some visual combinations of (diacritic + base) Latin letter. These double-byte encodings are precomposed in the sense that they represent a single character; there is no single-byte encoding for the diacritic character, and the representation of the second byte happens to be the same as that of the single-byte representation of the corresponding base character (e.g., 0xC14F represents *LATIN CAPITAL LETTER O WITH GRAVE* and 0xC16F represents *LATIN SMALL LETTER O WITH GRAVE*).

C90

The C90 Standard specified implementation-defined shift states rather than locale-specific shift states.

C++

The definition of multibyte character, 1.3.8, says nothing about encoding issues (other than that more than one byte may be used). The definition of multibyte strings, 17.3.2.1.3.2, requires the multibyte characters to begin and end in the initial shift state.

Common Implementations

Most methods for state-dependent encoding are based on ISO/IEC 2022:1994 (identical to the standard ECMA-35 "Character Code Structure and Extension Techniques", freely available from their Web site, http://www.ecma.ch). This uses a different structure than that specified in ISO/IEC 10646–1. The encoding method defined by ISO 2022 supports both 7-bit and 8-bit codes. It divides these codes up into control characters (known as *C0* and *C1*) and graphics characters (known as *G0*, *G1*, *G2*, and *G3*). In the initial shift state the *C0* and *G0* characters are in effect. ISO 2022

Table 243.1: Commonly seen ISO 2022 Control Characters. The alternative values for SS2 and SS3 are only available for 8-bit codes.

Name	Acronym	Code Value	Meaning
Escape	ESC	0x1b	Escape
Shift-In	SI	0x0f	Shift to the G0 set
Shift-Out	SO	0x0e	Shift to the G1 set
Locking-Shift 2	LS2	ESC 0x6e	Shift to the G2 set
Locking-Shift 3	LS3	ESC 0x6f	Shift to the G3 set
Single-Shift 2	SS2	ESC 0x4e, or 0x8e	Next character only is in G2
Single-Shift 3	SS3	ESC 0x4f, or 0x8f	Next character only is in G3

Some of the control codes and their values are listed in Table 243.1. The codes SI, SO, LS2, and LS3 are known as *locking shifts*. They cause a change of state that lasts until the next control code is encountered. A stream that uses locking shifts is said to use *stateful encoding*.

ISO 2022 specifies an encoding method: it does not specify what the values within the range used for graphic characters represent. This role is filled by other standards, such as ISO 8859. A C implementation that supports a state-dependent encoding chooses which character sets are available in each state that it supports (the C Standard only defines the character set for the initial shift state). 24 ISO 8859

Table 243.2: An implementation where G1 is ISO 8859–1, and G2 is ISO 8891–7 (Greek).

Encoded values	0x62	0x63	0x64	0x0e	0xe6	0x1b	0x6e	0xe1	0xe2	0xe3	0x0f
Control character				SO		LS2					SI
Graphic character	a	b	c		æ			α	β	γ	

Having to rely on implicit knowledge of what character set is intended to be used for G1, G2, and so on, is not always satisfactory. A method of specifying the character sets in the sequence of bytes is needed. The

ESC control code provides this functionality by using two or more following bytes to specify the character set (ISO maintains a *registry of coded character sets*). It is possible to change between character sets without any intervening characters. Table 243.3 lists some of the commonly used Japanese character sets.

C source code written by Japanese developers probably has the highest usage of shift sequences. There are several JIS (Japanese Industrial Standard) documents specifying representations for such sequences. Shift JIS (developed by Microsoft) belies its name and does not involve shift sequences that use a state-dependent encoding.

Table 243.3: ESC codes for some of the character sets used in Japanese.

Character Set	Byte Encoding	Visible Ascii Representation
JIS C 6226–1978	1B 24 40	<ESC> $ @
JIS X 0208–1983	1B 24 42	<ESC> $ B
JIS X 0208–1990	1B 26 40 1B 24 42	<ESC> & @ <ESC> $ B
JIS X 0212–1990	1B 24 28 44	<ESC> $ (D
JIS-Roman	1B 28 4A	<ESC> (J
Ascii	1B 28 42	<ESC> (B
Half width Katakana	1B 28 49	<ESC> (I

Table 243.4: A JIS encoding of the character sequence かな漢字(“kana and kanji”).

Encoded values	0x1b	0x24	0x42	0x242b	0x244a	0x3441	0x3b7a	0x1b	0x28	0x4a
Control character	<ESC>	$	B					<ESC>	(	J
Graphic character				か	な	漢	字			
Ascii characters				$+	$J	4A	;z			

Coding Guidelines

Developers do not need to remember the numerical values for extended characters. The editor, or program development environment, used to create the source code invariably looks after the details (generating any escape sequences and the appropriate byte values for the extended character selected by the developer). How these tools decide to encode multibyte character sequences is outside the scope of these coding guidelines.

It is usually possible to express an extended character in a minimal number of bytes using a particular state-dependent encoding. The extent to which developers might create fixed-length data structures on the assumption that multibyte characters will not contain any redundant shift sequences is outside the scope of this book. The value of the MB_LEN_MAX macro places an upper limit on the number of possible redundant shift sequences.

footnote 152 2017
MB_LEN_MAX 313

Example

```
1  #include <stdio.h>
2
3  char *p1 = "^[$B$3$l$OF|K\8lI=8=^[(J";  /* ^[$BF|K\8lJ8;zNs^[(J */
4  char *p2 = "^[$B$3$l$OF|1Q^[(Jmixed^[$BJ8;zNs^[(J";     /* Ascii + ^[$BF|K\8l^[(J */
5  char *p3 = "^[$B$3$l$OH>3Q^[(J^N6@6E^O^[$B$H^[(JASCII^[$B:.9g^[(J";
6
7  int main(void)
8  {
9  printf("%s^[$B$H^[(J%s^[$B$H^[(J%s\n", p1, p2, p3);
10 }
```

While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state.

Commentary

The implementation of a stateful encoding has to pick a special character, which is not in the basic character set, to indicate the start of a shift sequence. When not in the initial shift state, it is very unlikely that single bytes will be interpreted the same way as when in the initial shift state.

C++

The C++ Standard does not explicitly specify this requirement.

Common Implementations

The ESC character, 0x1b, is commonly used to indicate the start of a shift sequence.

---

245 12) The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set. footnote 12

Commentary

When trigraphs are used, it is possible to write C source code that contains only those characters that are in the Invariant Code Set of ISO/IEC 646.

C90

The C90 Standard explicitly referred to the 1983 version of ISO/IEC 646 standard.

---

246 The interpretation for subsequent bytes in the sequence is a function of the current shift state.

Commentary

This wording is really a suggestion for the design of multibyte shift states (it is effectively describing the processing performed by finite state machines, which is what a shift state encoding is). Being able to interpret a byte independent of the current shift state would indicate that the sequence of bytes that resulted in the current state were redundant.

The specification of the macro `MB_LEN_MAX` requires that the maximum number of bytes needed to handle a supported multibyte character be provided. It may, or may not, be possible to represent some redundant shift sequence within the available bytes. The standard does not explicitly require or prohibit support for redundant shift sequences. [313] MB_LEN_MAX

C++

A set of virtual functions for handling state-dependent encodings, during program execution, is discussed in Clause 22, Localization library. But, this requirement is not specified.

Common Implementations

Implementations usually use a simple finite state machine, often automatically generated, to handle the mapping of shift states into their execution character value. The extent to which sequences of redundant shift sequences is supported will depend on the implementation.

Coding Guidelines

The sequence of bytes in a shift sequence are usually generated via some automated process. For this reason a guideline recommending against the use of redundant shift sequences is unlikely to be enforceable, and none is given.

---

247 — A byte with all bits zero shall be interpreted as a null character independent of shift state. byte all bits zero

Commentary

This is a requirement on the implementation. This requirement makes it possible to search for the end of a string without needing any knowledge of the encoding that has been used. For instance, string-handling functions can copy multibyte characters without interpreting their contents.

C++

2.2p3 *. . . , plus a null character (respectively, null wide character), whose representation has all zero bits.*

While the C++ Standard does not rule out the possibility of all bits zero having another interpretation in other contexts, other requirements (17.3.2.1.3.1p1 and 17.3.2.1.3.2p1) restrict these other contexts, as do existing character set encodings.

---

multibyte character end in initial shift state

— ~~A byte with all bits zero shall not occur in the second or subsequent bytes of a~~ Such a byte shall not occur as part of any other multibyte character. 248

Commentary

This is a requirement on the implementation. The effect of this requirement is that partial multibyte characters cannot be created (otherwise the behavior is undefined). A null character can only exist outside of the sequence of bytes making up a multibyte character. For source files this requirement follows from the requirement to end in the initial shift state. During program execution this requirement means that library functions processing multibyte characters do not need to concern themselves with handling partial multibyte characters at the end of a string.

token 250 shift state

The wording was changed by the response to DR #278 (it is a requirement on the implementation that forbids a two-byte character from having a first, or any, byte that is zero).

C++

This requirement can be deduced from the definition of null terminated byte strings, 17.3.2.1.3.1p1, and null terminated multibyte strings, 17.3.2.1.3.2p1.

For source files, the following shall hold: 249

Commentary

These C-sentences specify requirements on a program. A program that violates them exhibits undefined behavior.

locale-specific behavior 44

implementation-defined behavior 42

Use of multibyte characters can involve locale-specific and implementation-defined behaviors. A source file does not affect the conformance status of any program built using it, provided its use of multibyte characters either involves locale-specific behavior or the implementation-defined behavior does not affect program output (e.g., they appear in comments).

Coding Guidelines

The creation of multibyte characters within source files is usually handled by an editor. The developer involvement in the process being the selection of the appropriate character. In such an environment the developer has no control over the byte sequences used. A guideline recommending against such usage is likely to be impractical to implement and none is given.

---

token shift state

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state. 250

Commentary

These are the only tokens that can meaningfully contain a multibyte character. A token containing a multibyte character should not affect the processing of subsequent tokens. Without this requirement a token that did not end in the initial shift state would be likely to affect the processing of subsequent tokens.

C90

Support for multibyte characters in identifiers is new in C99.

C++

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files. 116 translation phase 1

Coding Guidelines

The fact that many multibyte sequences are created automatically, by an editor, can make it very difficult for a developer to meet this requirement. A developer is unlikely to intentionally end a preprocessing token, created using a multibyte sequence, in other than the initial state. A coding guideline is unlikely to be of benefit.

---

251 — An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

Commentary

What is a valid multibyte character? This decision can only be made by a translator, should it chose to accept multibyte characters.

In C90 it was relatively easy to lexically process a source file containing multibyte characters. The context in which these characters occurred often meant that a lexer simply had to look for the character that terminated the kind of token being processed (unless that character occurred as part of a multibyte character).

Identifier tokens do not have a single termination character. This means that it is not possible to generalise support for multibyte characters in identifiers across all translators. It is possible that source containing a multibyte character identifier supported by one translator will cause another translator to issue a diagnostic.

C90

Support for multibyte characters in identifiers is new in C99.

C++

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files. 116 translation phase 1

Coding Guidelines

In some cases source files can contain multibyte characters and be translated by translators that have no knowledge of the structure of these multibyte characters. The developer is relying on the translator ignoring them in comments containing their native language, or simply copying the character sequence in a string literal into the program image. In other cases, for instance identifiers, knowledge of the encoding used for the multibyte character set is likely to be needed by a translator.

Ensuring that a translator capable of handling any multibyte characters occurring in the source is used, is a configuration-management issue that is outside the scope of these coding guidelines.

5.2.2 Character display semantics

Commentary

There is no guarantee that a character display will exist on any hosted implementation. If such a device is supported by an implementation, this clause specifies its attributes. character display semantics

C++

Clause 18 mentions "display as a wstring" in *Notes:*. But, there is no other mention of display semantics anywhere in the standard.

Common Implementations

Most Unix-based environments contain a database of terminal capabilities, the so-called *termcap* database.[1306] This database provides information to the host on a large number of terminal capabilities and characteristics. Knowing the display device currently being used (this usually relies on the user setting an environment variable) enables the database to be queried for device attribute information. This information can then be used by an application to handle its output to display devices. There is a similar database of information on printer characteristics. termcap database

The *active position* is that location on a display device where the next character output by the **fputc** function would appear. 252

Commentary

This defines the term *active position*; however, the term *current cursor position* is more commonly used by developers.

The wide character output functions act as if `fputc` is called.

C++

C++ has no concept of active position. The `fputc` function appears in "Table 94" as one of the functions supported by C++.

Other Languages

Most languages don't get involved in such low-level I/O details.

The intent of writing a printing character (as defined by the **isprint** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. 253

Commentary

The standard specifies an intent, not a requirement. Some devices produce output that cannot be erased later (e.g., printing to paper) while other devices always display the last character output at a given position (e.g., VDUs). The ability of printers to display two or more characters at the same position is sometimes required. For instance, programs wanting to display the **ô** character on a wide variety of printers might generate the sequence **o**, backspace, **^** (all of these characters are contained in the invariant subset of ISO 646).

The intended behavior describes the movement of the active position, not the width of the character displayed. There is nothing in this definition to prevent the writing of one character affecting previously written characters (which can occur in Arabic). This specification implies that the positions are a fixed width apart.

glyph 58

The graphic representation of a character is known as a *glyph*.

C++

The C++ Standard does not discuss character display semantics.

Common Implementations

In some oriental languages, character glyphs can usually be organized into two groups, one being twice the width as the other. Implementations in these environments often use a fixed width for each glyph, creating empty spaces between some glyph pairs.

Some orthographies, which use an alphabetic representation, contain single characters that use what appears to be two characters in their visual representation. For instance, the character denoted by the Unicode value U00C6 is *Æ*, and the character denoted by the Unicode value U01C9 is *lj*. Both representations are considered to be a single character (the former is also a single letter, while the latter is two letters).

Coding Guidelines

The concept of active position is useful for describing the basic set of operations supported by the C Standard. The applications' requirements for displaying characters may, or may not, be feasible within the functionality provided by the standard; this is a top-level application design issue. How characters appear on a display device is an application user interface issue that is outside the scope of this book.

writing direction
locale-specific

The direction of writing is locale-specific. 254

Commentary

Although left-to-right is used by many languages, this direction is not the only one used. Arabic uses right-to-left (also Hebrew, Urdu, and Berber). In Japanese it is possible for the direction to be from top to bottom with the lines going right-to-left (mainland Chinese has the columns going from left-to-right, in Taiwan it goes right-to-left), or left-to-right with the lines going top to bottom (the same directional conventions as English)

There is no requirement that the direction of writing always be the same direction, for instance, braille alternates in direction between adjacent lines (known as *boustrophedron*), as do Egyptian hieroglyphs, Mayan, and Hittite. Some Egyptian hieroglyphic characters can face either to the left or right (e.g.,  or ), information that readers can use to deduce the direction in which a line should be read.

Some applications need to simultaneously handle locales where the direction of writing is different, for instance, a word processor that supports the use of Hebrew and English in the same document. This level of support is outside the scope of the C Standard.

C++

The C++ Standard does not discuss character display semantics.

Coding Guidelines

The direction of writing is an application issue. Any developer who is concerned with the direction of writing will, of necessity, require a deeper involvement with this topic than the material covered by the C Standard or these coding guidelines.

Example

The direction of writing can change during program execution. For instance, in a word processor that handles both English and Arabic or Hebrew, the character sequence *ABCdefGHJ* (using lowercase to represent English and uppercase to represent Arabic/Hebrew) might appear on the display as **JHGdefCBA**.

---

255 If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

Commentary

The Committee recognized that there is no commonality of behavior exhibited by existing display devices when the final position on a line is reached.

C++

The C++ Standard does not discuss character display semantics.

Common Implementations

Some display devices wrap onto the next line, effectively generating an extra new-line character. Other devices write all subsequent characters, up to the next new-line character, at the final position. On some displays, writing to the bottom right corner of a display has an effect other than displaying the character output, for instance, clearing the screen or causing it to scroll. The `termcap` and `ncurses` both provide configuration options that specify whether writing to this display location has the desired effect.

Coding Guidelines

Organizing the characters on a display device is an application domain issue. The fact that the C Standard does not provide a defined method of handling the situation described here needs to be dealt with, if applicable, during the design process. This is outside the scope of these coding guidelines.

---

256 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:

Commentary

This is the behavior of Ascii terminals enshrined in the C Standard.

Rationale To avoid the issue of whether an implementation conforms if it cannot properly effect vertical tabs (for instance), the Standard emphasizes that the semantics merely describe *intent*.

These escape sequences can also be output to files. The data values written to a file may depend on whether the stream was opened in text or binary mode.

C++

The C++ Standard does not discuss character display semantics.

Other Languages

Java provides a similar set of functionality to that described here.

Common Implementations

Most display devices are capable of handling most of the functions described here.

Coding Guidelines

A program cannot assume that any of the functionality described will occur when the escape sequence is sent to a display device. The root cause for the variability in support for the intended behaviors is the variability of the display devices. In most cases an implementation's action is to send the binary representation of the escape sequence to the device. The manufacturers of display devices are aware of their customers expectations of behavior when these kinds of values are received.

There is little that coding guidelines can recommend to help reduce the dependency on display devices. The design guidelines of creating individual functions to perform specific operations on display devices and isolating variable implementation behaviors in one place are outside the scope of these coding guidelines.

\a (*alert*) Produces an audible or visible alert without changing the active position. 257

Commentary

The intent of an alert is to draw attention to some important event, such as a warning message that the host is to be shut down, or that some unexpected situation has occurred. A program running as a background process (a concept that is not defined by the C Standard) may not have a display device attached (does a tree falling in a forest with nobody to hear it make a noise?).

C++

Alert appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

Common Implementations

Most implementations provide an audible alert. On display devices that don't have a mechanism for producing a sound, a visible alert might be to temporarily blank the screen or to temporarily increase the brightness of the screen.

Coding Guidelines

Programs that produce too many alerts run the risk of having them ignored. The human factor involved in producing alerts are outside of the scope of these coding guidelines. Issues such as a display device not being able to produce an audible alert because its speaker is broken, is also outside the scope of these coding guidelines.

---

backspace escape sequence

\b (*backspace*) Moves the active position to the previous position on the current line. 258

Commentary

The standard specifies that the active position is moved. It says nothing about what might happen to any character displayed prior to the backspace at the new current active position.

Common Implementations

Some devices erase any character displayed at the previous position.

C++

Backspace appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12: 17.4.1.2p3

Example

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  printf("h\bHello \b World\n");
6  }
```

---

259 If the active position is at the initial position of a line, the behavior of the display device is unspecified.

Commentary

Some terminals have input locking states. In such cases an unspecified behavior put the display device into a state where it no longer displays characters written to it.

C90

If the active position is at the initial position of a line, the behavior is unspecified.

This wording differs from C99 in that it renders the behavior of the program as unspecified. The program simply writes the character; how the device handles the character is beyond its control.

C++

The C++ Standard does not discuss character display semantics.

Common Implementations

The most common implementation behavior is to ignore the request leaving the active position unchanged. Some VDUs have the ability to wrap back to the final position on the preceding line.

Coding Guidelines

While it may be technically correct to specify that the behavior of the display device as unspecified, it does indirectly affect the output behavior of a program in that subsequent output may not appear on that display device.

---

260 `\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page.

Commentary

Whatever a page, logical or otherwise, is. This concept is primarily applied to printers. The functionality to move to the start of the next page, from anywhere on the current page, is generally provided by printer vendors. Programs might use this functionality since it frees them from needing to know the number of lines on a page (provided the minimum needed to support the generated output is available).

page
logical

C++

Form feed appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

Coding Guidelines

Use of this escape sequence could remove the need for a program to be aware of the number of lines on the page of the display device being written. However, it does place a dependency on the characteristics of the display device being known to the host executing the program, or on the device itself, to respond to the data sent to it.

termcap database

---

new-line escape sequence

\n (*new line*) Moves the active position to the initial position of the next line. 261

Commentary

What happens to the preceding lines is not specified. For instance, whether the display device scrolls lines or wraps back to the top of any screen. The standard is silent on the issue of display devices that only support one line. For instance, do the contents of the previous line disappear?

C++

New line appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

Other Languages

Some languages provide a library function that produces the same effect.

Common Implementations

On some hosts the new-line character causes more than one character to be sent to the display device (e.g., carriage return, line feed).

A printing device may simply move the media being printed on. A VDU may display characters on some previous line (wrapping to the start of the screen). On some display devices (usually memory-mapped ones), the start of a new line is usually indicated by an end-of-line character appearing at the end of the previous line. On other display devices, a fixed amount of storage is allocated for the characters that may occur on each line. In this case the end of line is not stored as a character in the display device.

end-of-line representation 224

Coding Guidelines

Issues, such as handling lines that are lost when a new line is written or display devices that contain a single line, are outside the scope of these coding guidelines.

---

carriage return escape sequence

\r (*carriage return*) Moves the active position to the initial position of the current line. 262

Commentary

The behavior might be viewed as having the same effect as writing the appropriate number of backspace characters. However, the effect of writing a backspace character might be to erase the previous character, while a carriage return does not cause the contents of a line to be erased. Like backspace, the standard says nothing about the effect of writing characters at the position on a line that has previously been written to.

backspace escape sequence 258

C++

Carriage return appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12: 17.4.1.2p3

---

263 `\t` (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. horizontal tab escape sequence

Commentary

Horizontal tabulation positions are provided by vendors of display devices as a convenient method of aligning data, on different lines, into columns. In some cases they can remove the need for a program to count the number of characters that have been written. The C Standard does not provide a method for controlling the location of horizontal tabulation positions. Neither does a program have any method of finding out which positions they occupy.

C++

Horizontal tab appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12: 17.4.1.2p3

Common Implementations

The location of tabulation positions on a line are usually controlled by the display device. There may be a limited number that can be configured on a line. Configuring a horizontal tab position every eight active positions from the start of the line is a common default. Many hosts allow the default setting to be changed, and some users actively make use of this configuration option.

Coding Guidelines

A commonly seen application problem is the assumption, by the developer, of where the horizontal tabulation positions occur on a display device. However, the handling display devices are outside the scope of these coding guidelines.

---

264 If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.

Commentary

The standard does not specify how many horizontal tabulation positions must be supported by an implementation, if any.

C90

If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified.

Common Implementations

Some implementations do not move the active position when the last defined horizontal tabulation position has been reached; others treat writing such a character as being equivalent to writing a single white-space character at this position. In some cases the behavior is to move the active position to the first horizontal tabulation position on the next line.

vertical tab escape sequence

\v (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. 265

Commentary

Although the standard recognizes that the direction of writing is locale-specific, it says nothing about the order in which lines are organized. The vertical tab (and new line) escape sequence move the active position in the same line direction. There is no escape sequence for moving the active position in the opposite direction, similar to backspace for movement within a line.

page 260 logical

The concept of vertical tabulation implicitly invokes the concept of *current page*. This concept is primarily applied to printers, while the dimensions of a page might be less variable than a terminal. Before laser printers were invented, it was very important to ensure that output occurred in a controlled, top-down fashion.

C++

Vertical tab appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

Common Implementations

In most implementations a vertical tab moves the active position to the next line, with the relative position within the line staying the same.

If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified. 266

Commentary

The intended behavior is likely to vary between terminals and printers.

C90

If the active position is at or past the last defined vertical tabulation position, the behavior is unspecified.

Common Implementations

Many display devices do not define vertical tabulation positions; this escape sequence simply causes the active position to move to the next line. The behavior is the same as when a new line escape sequence is written at the end of a page, or screen.

---

escape sequence fit in char object

Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single **char** object. 267

Commentary

basic execution character set 221
basic character set fit in a byte 222

These escape sequences are defined to be members of the basic execution character set and also to fit in a byte.

The mapping to this implementation-defined value occurs at translation time. The execution time value actually received by the display device is outside the scope of the standard. The library function `fputc` could map the value represented by these single **char** object into any sequence of bytes necessary.

C++

This requirement can be deduced from 2.2p3.

Other Languages

Java explicitly defines the values of the escape sequences it specifies.

Common Implementations

The specified escape sequences are available in the Ascii character set (and thus also in ISO 10646). 28 ISO 10646

268 The external representations in a text file need not be identical to the internal representations, and are outside the scope of this International Standard.

Commentary

The Committee recognizes that host file systems may use a representation for text files that is different from that used for binary files. The output functions will know the mode with which a stream was opened and can process the bytes written appropriately. There is a guarantee for binary files, which does not hold for text files, that the bytes written out shall compare equal to the same bytes read back in again.

C++

The C++ Standard does not get involved in such details.

Common Implementations

The external representation of a text file is usually the same as that used to hold a C source file. The representation of the new line escape sequence is usually the same as that for end-of-line, which is not always a single character. 224 end-of-line representation

From an executing program's point of view, on hosts that support output redirection, there may be no distinction made between a display device and a text file. However, the driver for a display device may respond differently for some characters.

269 **Forward references:** the **isprint** function (7.4.1.8), the **fputc** function (7.19.7.3).

5.2.3 Signals and interrupts

270 **Commentary**

signal

Signals are difficult to specify in a system-independent way. The C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a volatile static variable which can be written uninterruptedly and promptly return. Rationale

. . .

A second signal for the same handler could occur before the first is processed, and the Standard makes no guarantees as to what happens to the second signal.

A pole exception is the same as a divide-by-zero exception: a finite non-zero floating-point number divided by a zero floating-point number. WG14/N748

Currently, various standards define the following exceptions for the indicated sample floating-point operations. For LIA–2, there are other operations that produce the same exceptions.

LIA	<---------- Standard ---------->			IEEE
Exception	LIA-1	LIA-2	IEEE-754/IEC-559	Exception
undefined	0.0 / 0.0	sqrt(-1.0)	0.0 / 0.0	invalid
	1.0 / 0.0	log(-1.0)	infinity / infinity	
			infinity - infinity	
			0.0 * infinity	

			sqrt(-1.0)	
pole	(not yet)	log(0.0)	1.0 / 0.0	division by zero
floating_ overflow	max * max	exp(max)	max * max	overflow
	max / min		max / min	
	max + max		max + max	
underflow	min * min	exp(-max)	min * min	underflow
	min / max		min / max	

In the above table, 1.0/0.0 is a shorthand notation for any non-zero finite floating-point number divided by a zero floating-point number; max is the maximum floating-point number (FLT_MAX, DBL_MAX, LDBL_MAX); min is the minimum floating-point number (FLT_MIN, DBL_MIN, LDBL_MIN); log() and exp() are mathematical library routines.

We believe that LIA–1 should be revised to match LIA-2, IEC-559 and IEEE-754 in that 1.0/0.0 should be a pole exception and 0.0/0.0 should be an undefined exception.

C++

The C++ Standard specifies, Clause 15 Exception handling, a much richer set of functionality for dealing with exceptional behaviors. While it does not go into the details contained in this C subclause, they are likely, of necessity, to be followed by a C++ implementation.

Other Languages

Some languages (e.g., Ada, Java, and PL/1) define statements that can be used to control how exceptions and signals are to be handled. After over 30 years floating point exception handling has finally been specified in the Fortran Standard.[651] A few languages include functionality for handling signals and interrupts, but most ignore these issues.

Common Implementations

Implementations are completely at the mercy of what signals are supported by the host environment and what interrupts are generated by the processor. Gould (Encore) PowerNode treated both floating-point and integer overflow as being the same.

Coding Guidelines

This subclause lists those minimum characteristics of a program image needed to support signals and interrupts. Such support by the implementations is only half of the story. A program that makes use of signals has to organize its behavior appropriately. Techniques for writing programs to handle signals, or even ensuring that they are thread-safe are outside the scope of these coding guidelines.

Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. 271

Commentary

This is a requirement on the implementation. An implementation may provide a mechanism for the developer to switch off interrupts within time-critical functions. Although such usage is an extension to the standard, it cannot be detected in a strictly conforming program.

How could an implementation's conformance to this requirement be measured? A program running under an implementation that supports some form of external interrupt, for instance SIGINT, might be executed a large number of times, the signal handler recording where the program was interrupted (this would require functionality not defined in the standard). Given sufficient measurements, a statistical argument could be used to show that an implementation did not support this requirement. A nonprogrammatic approach would be to verify the requirement by understanding how the generated machine code interacted with the host processor and the characteristics of that processor.

This wording is not as restrictive on the implementation as it first looks. The only signal that an implementation is required to support is the one caused by a call to the `raise` function. Requiring that any developer-written functions be callable from a signal handler restricts the calling conventions that may be used in such a handler to be compatible with the general conventions used by an implementation. This simplifies the implementation, but places a burden on time-critical applications where the calling overhead may be excessive.

C++

This implementation requirement is not specified in the C++ Standard (1.9p9).

Other Languages

Most languages don't explicitly say anything about the interruptibility of a function.

Common Implementations

Few if any host processors allow execution of instructions to be interrupted. The boundary at the completion of one instruction and starting another is where interrupts are usually responded to. In the case of pipelined processors, there are two commonly seen behaviors. Some processors wait until the instructions currently in the pipeline have completed execution, while others flush the instructions currently in the pipeline. An example of an instruction that causes an interrupt to be raised after it has only partially completed is one that accesses storage, if the access causes a page fault (causing the instruction to be suspended while the accessed page is swapped into storage). Another case is performing an access to storage using a misaligned address, or an invalid address. In these cases the instruction may never successfully complete.

External, nonprocessor-based interrupts are usually only processed once execution of the current instruction is complete. Some processors have instructions that can take a relatively long time to execute, for instance, instructions that copy large numbers of bytes between two blocks of memory. Depending on the design requirements on interrupt latency, some processors allow these instructions to be interrupted, while others do not.

Some implementations[1340] require that functions called by a signal handler preserve information about the state of the execution environment, such as register contents. Developers are required to specify (often by using a keyword in the declaration, such as **`interrupt`**) which functions must save (and restore on return) this information.

---

272 All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis.

object storage outside function image

Commentary

This is a requirement on the implementation (although the as-if rule might be invoked). The model being described is effectively a stack-based approach to the calling of functions and the handling of storage for objects they define (the actual storage allocation could use a real stack or simulate one using allocated storage).

Storing objects in the function image, or simply having a preallocated area of storage for them, would prevent a function from being called recursively (having more than one call to a function in the process of being executed at the same time is a recursive invocation, however the invocation occurred). An implementation is required to support recursive function calls. This requirement prevents implementations using a technique that was once commonly used (primarily by implementations of other languages), but can have different execution time semantics when recursive calls are made.

1026 function call recursive

C++

The C++ Standard does not contain this requirement.

Other Languages

Most languages require support for recursive function calls, implying this requirement.

1026 function call recursive

Common Implementations

Modern processors try to separate code (function image) and data (object definitions). Accesses to the two have different characteristics, which affects the design of caches for them (often implemented as two separate cache areas on the chip). Independently of processor support, the host environment (operating system) may mark certain areas of storage as having execute-only permission. Attempts to read or write to such storage, from an executing program, often leads to a signal being raised.

Applications targeted at a freestanding environment rarely involve recursive function calls. Storage may also be at a premium and hardware stack support limited (the Intel 8051[625] is limited to a 128-byte stack). Some hosts allocate fixed areas, in static storage, for objects local to functions. A call tree, built at link-time, can be used to work out which storage areas can be shared by overlaying those objects whose lifetimes do not overlap, reducing the fixed execution time memory overhead associated with such a design.

Many processors have span-dependent load and store instructions. That is, a short-form (measured in number of bytes) that can only load (or store) from/to storage locations whose address has a small offset relative to a base address, while a long-form supports larger offsets. When storage usage needs to be minimized, it may be possible to use a short-form instruction to access storage locations in the function image. The usual technique used is to reserve storage for objects after an unconditional branch instruction, which is accessed by the instructions close (within the range supported by the short-form instruction) to those locations.[1174]

Coding Guidelines

While implementations might be required to allocate objects outside of a function image, developers have been known to write code to store values in a program image. In those few cases where values are stored in this way, the developers involved are very aware of what they are doing. A guideline recommendation serves no purpose.

Example

The following is one possible method that might be used to store data in a program image.

```
#include <stdio.h>

extern int always_zero = 0;
static int *code_ptr;

void f(void)
{
/*
 * No static object declarations in this function ;-)
 */
if (always_zero == 1) /* create some dead code */
   {
   /*
    * Pad out with enough code to create storage for an int.
    * A smart optimizer is the last thing we need here.
    */
   always_zero++;
   always_zero++;
   }

(*code_ptr)++;
printf("This function has been called %d times.\n", *code_ptr);
}

void init(void)
{
/*
 * The value 16 is the offset of the dead code from the start of the
 * function.  Change to suit your local instruction sizes (this works
```

```
 * for gcc on an Intel x86).  We also need to make sure that the
 * pointer to int is correctly aligned.  A reliable guess is that
 * the alignment is a multiple of the object size.
 */
code_ptr=(int *)((((int)(char *)f) + 16) & ~(sizeof(int)-1));
*code_ptr=0;
}

int main(void)
{
init();
for (int index=0; index < 10; index++)
   f();
}
```

5.2.4 Environmental limits

273 Both the translation and execution environments constrain the implementation of language translators and libraries.

environmental limits

Commentary

In the past the available storage capacity of the translation environment has been an implementation design consideration for translator vendors. Translating programs containing large numbers of some constructs sometimes exceeded the available host capacity. This is rarely the case today.

In some environments, particularly freestanding ones, there can be severe constraints on the execution environment.

C++

There is an informative annex which states:

> *Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process.*

Annex Bp1

Common Implementations

While it might be theoretically possible to create a translator that is not affected by its own environment, the cost is likely to outweigh the benefits.

Coding Guidelines

Some limits, imposed by the translation environment, are best dealt with as they are encountered. The cost of structuring a program to deal with all lowest common denominators is rarely recouped in future savings (in reduced porting costs). Programs are often targeted at a particular class of environments (e.g., workstations or hand-held devices). Execution time constraints can have a large impact and may affect the choice of algorithms as well as how the source is structured. Both of these issues are dealt with, by these coding guidelines, as they are encountered in the C Standard wording.

274 The following summarizes the language-related environmental limits on a conforming implementation;

Commentary

The intent is that these are base limits and commercial pressure will encourage vendors to create implementations that improve on them. By specifying such limits the Committee is providing a guide as to what can be expected, by a developer, of an implementation.

C++

There is an informative annex which states:

Annex Bp2 *The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine conformance.*

Other Languages

Most language standards are silent on the subject of environmental limits and provide no guide on the number of constructs that a translator might be expected to handle. The Modula-2 Standard specifies minimum translator limits for many of the language constructs covered by the C Standard (Pronk[1127] tests the minimum values supported by a number of translators).

Common Implementations

Most translators allocate space for the symbol table and other information, dynamically as the source code is processed. This choice of implementation technique does not remove the limit on the total amount of memory available to a translator, but it does provide flexibility. There are a few cases where limits may be imposed because an implementation has chosen to use fixed-size data structures.

macro replacement

One limit not mentioned in the standard is the maximum number of characters in a macro, during expansion. Several implementations have limits in this area, sometimes as low as 256. The limit for macro definitions is covered by the logical line length.

limit characters on line 292

Coding Guidelines

Exceeding any defined minimum limits is a calculated risk. Some of the limits may be hard to design around; for instance, the number of identifiers with external linkage. What is the cost, both in developer effort and loss of design integrity, of adapting a program to fit within these limits? What is the likelihood of encountering a translator that cannot process a source file that exceeds some limit? Is it worth paying the cost to be certain of having source that is translatable by such translators? While the environments where translators are resource-limited are becoming rare, many translators continue to contain some of their own, internal, fixed limits.

A translator may have other limits that are not described in the C Standard. These will have to be dealt with, by developers, as they are encountered.

---

the library-related limits are discussed in clause 7. 275

C++

Clause 18.2 contains an *Implementation Limits:*.

5.2.4.1 Translation limits

---

translation limits

The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:[13)] 276

Commentary

This is a requirement on the implementation (a single preprocessing translation unit containing all of the constructs given here, to the limits specified). The topic of a perverse implementation, one that can successfully translate a single program containing all of these limits but no other program, crops up from time to time. Although of theoretical interest, this discussion is of little practical interest, because writing a translator that only handled a single program would probably require more effort than writing one that handled programs in general.

The values for these limits were not obtained by measuring how often each construct appeared within existing source code. There is no claim that a program containing an instance of all such constructs is in any way representative of a typical program.

Some of the limits chosen represent interesting compromises. The goal was to allow reasonably large portable programs to be written, without placing excessive burdens on reasonably small implementations, some of which might run on machines with only 64 K of memory. In C99, the minimum amount of memory for the target machine was raised to 512 K. In addition, the Committee recognized that smaller machines rarely serve as a host for a C compiler: programs for embedded systems or small machines are almost always developed using a cross compiler running on a personal computer or workstation. This allows for a great increase in some of the translation limits.

A program containing an instance of all such limits is one of the tests included in the commercially available C validation suites that used to be used by NIST and BSI.

C++

However, these quantities are only guidelines and do not determine conformance. Annex Bp2

This wording appears in an informative annex, which itself has no formal status.

Other Languages

Many language definitions specify some minimum value for some constructs that implementations are required to support. The Modula-2 Standard contains what it called *limit-specification generators*. These are a set of Modula-2 programs, which when executed generate a set of Modula-2 programs that an implementation must be capable of translating and executing.

Common Implementations

Some implementations provide a translator option that allows the developer to control the amount of storage allocated to various internal data structures; for instance, the option `-xs1234` might specify a symbol table capable of holding 1,234 symbols. For large programs it can take several attempts before the various options are tuned (enabling the source to be translated within the available storage). Such implementation options may still be provided today, as part of a backwards compatibility mode.

Coding Guidelines

Most of the limit values are sufficiently generous that few of them are likely to be exceeded. But within these coding guidelines, we are not just interested in translator limitations, we are also interested in developer limitations. There may be readability, comprehensibility, or complexity issues associated with multiple occurrences of some constructs. A program that contains an excessive number of any particular construct could be poorly structured or simply a large program.

In the case of nested constructs, it is often claimed that developers have problems remembering the information if the nesting is too deep. The fact that developers experience problems remembering information on nested constructs suggests they are using short-term memory to hold this information. The capacity limits of short-term memory are only one of the issues involved in comprehending nested constructs. How developers organize information presented to them (from the source code), knowledge held in their long-term memories (about how a program works or memories of previous code readings), and the extent to which information from different nesting levels is related all need to be considered.

0 memory developer

1821 coupling and cohesion

It is also important to consider the bigger picture of particular nested constructs. Are the alternatives any better? Some coding guideline documents specify a value for the maximum nesting level of particular constructs[941,942] (sometimes giving a rationale based on the famous 7±2 paper). These coding guidelines resist the attractions of providing a single, easy-to-calculate, maximum nesting limit. The issues are discussed in more detail within each nested construct.

0 Miller 7±2

277— 127 nesting levels of blocks

limit block nesting

Commentary

block 1710 compound 1729 statement syntax

Blocks are created by a number of different kinds of statements. The one most commonly thought of is a compound statement. There is no dependency on the kinds of statement that cause a block to be created. There could be any combination of **`if`**/**`while`**/**`for`**/**`switch`** or simply a compound statement with no associated header.

This limit might be reached in automatically generated code, but it would be considered an extreme case.

C90

15 nesting levels of compound statements, iteration control structures, and selection control structures

block 1741 selection statement block 1742 selection sub-statement block 1768 iteration statement block 1769 loop body

The number of constructs that could create a block increased between C90 and C99, including selection statements and their associated substatements, and iteration statements and their associated bodies. Although use of these constructs doubles the number of blocks created in C99, the limit on the nesting of blocks has increased by a factor of four. So, the conformance status of a program will not be adversely affected.

C++

The following is a non-normative specification.

Annex Bp2 *Nesting levels of compound statements, iteration control structures, and selection control structures [256]*

Common Implementations

Nesting of blocks is part of the language syntax and is usually implemented with a table-driven syntax analyzer. Table-driven syntax analyzers maintain their own stack, often a predefined fixed size, of information. A very large number of nested blocks is likely to cause this parser table to overflow.

Coding Guidelines

In human-written code a significantly lower limit on the nesting of blocks is often recommended. Working purely on the basis of some form of line indentation, for every new block opened, more than five nested levels would lead to a visually difficult to follow, on a display device, source file. Blocks opened and closed within a macro definition would not affect the visual appearance of source, at the point of macro invocation. This kind of nesting would not be counted in the five recommendation.

— 63 nesting levels of conditional inclusion 278

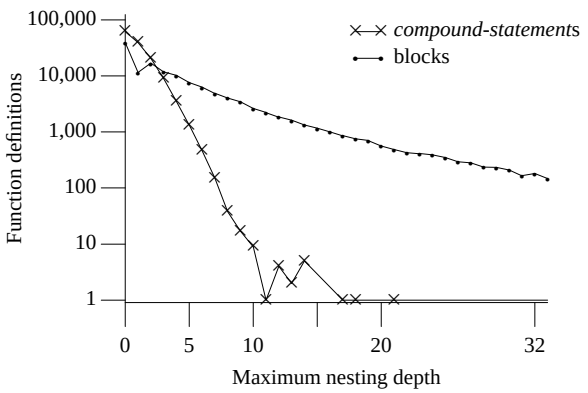


Figure 277.1: Number of functions containing blocks and *`compound-statement`*s nested to the given maximum nesting level. Based on the visible form of the `.c` files.

Commentary

Conditional inclusion is performed as part of preprocessing. As such, it is independent of the syntax processing performed by subsequent translation phases and is given its own limit. conditional inclusion

The value of this limit is consistent with other limit values. It is something of a fortunate coincidence, because the same ratios applied in C90, where the following rationale did not apply. The value is half the limit value for nesting of blocks. This difference occurs because the C **if** statement is defined to create two blocks. Nesting **if** statements 64 deep would be sufficient to exceed the block limit, and 64 nested **#if** directives would exceed the above limit. 277 limit block nesting 1741 block selection statement

C90

8 nesting levels of conditional inclusion

C++

The following is a non-normative specification.

Nesting levels of conditional inclusion [256] Annex Bp2

Common Implementations

Several different techniques are used to write preprocessors. Given the relatively simple C preprocessor syntax, many have a *flat* view of the nesting of these constructs. They simply maintain a count of the current nesting level. A full parser/syntax-based approach faces the same type of problems found in handling the C syntax limits discussed here. But, such an approach is rarely seen in C preprocessor implementations.

This limit may be reached in automatically generated code.

Coding Guidelines

Conditional inclusion differs from selection statements in that it is not possible to prevent deep nesting by moving directives to separate functions (although they could be moved to a separate source file and accessed via a **#include** directive, or the design of the configuration control implied by the nested directives could be changed). 1739 selection statement syntax

The human factors issues might be thought to be the same as those for the nesting of selection statements. However, developers generally do not visually indent nested conditional inclusion directives (see Figure 1854.1) a practice that is commonly used for selection (and other) statements.

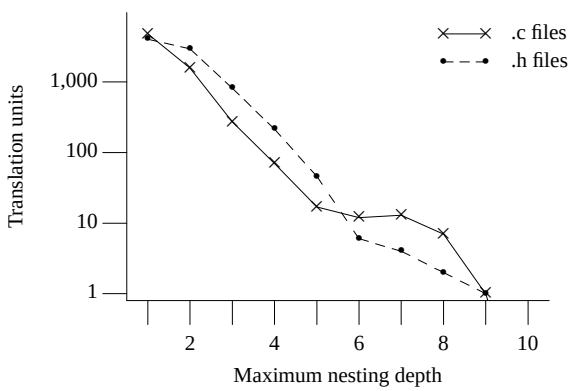


Figure 278.1: Number of translation units containing conditional inclusion directives nested to the given maximum nesting level. Based on the visible form of the `.c` and `.h` files.

limit type complexity

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration 279

Commentary

This limit is based on a particular encoding of type information within 32 bits, which was used in some early translators. Eight bits were used to encode the base type and up to 12 lots of 2 bits representing pointer-to, array-of, or function-of. Another consideration was that Fortran originally supported a maximum of seven subscripts in an array declaration. The committee recognized that Fortran to C translations needed to be able to support this number of nested array declarations.

limit 279 type complexity

Wording that appears elsewhere specifies that types defined via typedef names need to be included in the count.

C++

The following is a non-normative specification.

Annex Bp2

Pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration [256]

Common Implementations

Some implementations continue to use the K&R technique. Many others use a dynamic data structure relevant to the type being defined and have no internal limits on the complexity supported.

Coding Guidelines

Data structures need to mimic the application domain being addressed. If a deep nesting of pointers, arrays, or function declarators is called for, there may be little benefit in arbitrarily splitting the declaration into smaller components, unless these subcomponents have semantic meaning within the application domain.

Example

```
1  char ************p_12;
2  short a_12[1][1][1][1][1][1][1][1][1][1][12];
3  void (* (* (* (* (* (* (* (* (* (* (*p)
4     (void))(void))(void))(void))(void))(void))(void))(void))(void))(void))(void);
```

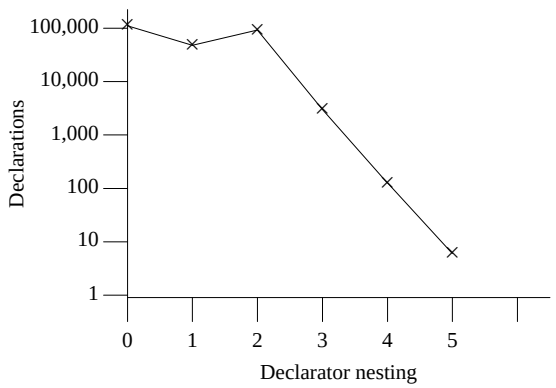


Figure 279.1: Number of full declarators containing a given number of modifiers. Based on the translated form of this book's benchmark programs.

280 — 63 nesting levels of parenthesized declarators within a full declarator

limit declarator parentheses

Commentary

The limit of 12 modifiers on a declaration is likely to be reached before this limit of 63 is reached on a full declarator (unless redundant ( ) are used, or some very rarely seen structure declarations). This limit is unlikely to be reached, even in automatically generated code. 1549 full declarator

```
1  struct {
2          (struct {
3                  (struct {
4                          ...
5                  } *p)[2];
6          } *q)[1];
7  };
```

C90

31 nesting levels of parenthesized declarators within a full declarator

C++

The C++ Standard does not discuss declarator parentheses nesting limits.

Common Implementations

Nesting of parentheses is part of the language syntax and is usually implemented with a table-driven syntax analyzer. The same implementation details as nesting of blocks applies. The nesting of parentheses can occur within a nested block, slightly increasing the chances of reaching an internal implementation limit. 277 limit block nesting

281 — 63 nesting levels of parenthesized expressions within a full expression

parenthesized expression nesting levels

Commentary

While it is possible to keep within this limit in an expression containing one instance of every operator (C contains 47 unique operators), an expression containing more than one instance of two operators may need to exceed this limit— for instance, `(((((a0/x+a1)/x+a2)/x+a3)/x+a4)/x+a5)/x+...`.

This limit is rarely reached except in automatically generated code. Even then it is rare.

C90

31 nesting levels of parenthesized expressions within a full expression

C++

The following is a non-normative specification.

Nesting levels of parenthesized expressions within a full expression [256] Annex Bp2

Common Implementations

The same implementation details as nesting of declarators applies, with the difference that nesting of expressions is more common and likely to be deeper. 280 limit declarator parentheses

Coding Guidelines

memory 0 chunking

Although some uses of parentheses may be technically redundant, they may be used to simplify the visual appearance of an expression, or to divide an expression into meaningful chunks. While minimizing the number of parentheses in an expression may be an interesting mathematical problem, minimization is not a desirable goal when writing source code. The top priority when considering the use of parentheses should always be comprehensibility of the resulting expression.

```
1  (((((a0 * x + a1) * x + a2) * x + a3) * x + a4) * x + a5) * x + a6
2
3  a * (b + (c << (d > (e == (f & (g ^ (h | (i && (j || (k_1 ? k_2 : (L = (m , n ))))))))))));
```

expressions 940 The issue of expression complexity is discussed elsewhere.

---

internal identifier significant characters

— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character) 282

Commentary

An internal identifier is one whose name is never visible outside of the source file in which it is declared. Because it is not necessary to worry about external representation issues, it is possible to count one UCN as one character.

significant 2036 characters future language directions

This limit may be reached in automatically generated code.

This minimum limit may be increased in a future revision of the standard.

C90

31 significant initial characters in an internal identifier or a macro name

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-normative specification.

Annex Bp2

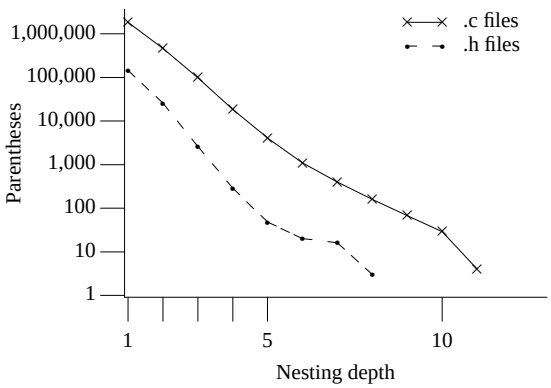


Figure 281.1: Nesting of all occurrences of parentheses. Based on the visible form of the `.c` and `.h` files.

Number of initial characters in an internal identifier or a macro name [1024]

Other Languages

Some languages are silent on the number of significant characters in an internal identifier; others specify the same limit as external identifiers.

Common Implementations

This is one area where translators are likely to use a fixed-size data structure (usually an array). Using a linked list of characters to represent an identifier name would be a significant overhead. Having a fixed-size data structure that grows once the available free space is filled is an alternative used by some implementations.

Coding Guidelines

The issue of identifier length is discussed elsewhere.

792 identifier number of characters

Usage

Very few identifiers approach the C99 translation limit (see Figure 792.7).

---

283 — 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[14)]

external identifier significant characters

Commentary

Information on externally visible identifiers needs to be stored in the files (usually object files) created by a translator. This information is compared against identifiers declared in other translation units when linking to build a program image. The predefined format of such files (not always within the control of the translator writer) may have limitations on what characters are acceptable in an identifier.

141 program image

The values of 6 and 10 were chosen so that the encodings *\u1234* and *\U12345678* could be used.

C90

6 significant initial characters in an external identifier

C++

All characters are significant.[20)]

2.10p1

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-Normative specification.

Number of initial characters in an external identifier [1024]

Annex Bp2

Other Languages

The Fortran significant character limit of six was followed by many suppliers of linkers for a long time. The need for longer identifiers to support name mangling in C++ ensured that most modern linkers support many more significant characters in an external identifier.

Common Implementations

Historically, the number of significant characters in an external identifier was driven by the behavior of the host vendor-supplied linker. Only since the success of MS-DOS have developers become used to translator vendors supplying their own linker. Previously, most linkers tended to be supplied by the hardware vendor.

The mainframe world tended to be driven by the requirements of Fortran, which had six significant characters in an internal or external identifier. In this environment it was not always possible to replace the system linker by one supporting more significant characters. The importance of the mainframe environment waned in the 1990s. In modern environments it is very often possible to obtain alternative linkers.

Coding Guidelines

The number of significant characters should not affect the choice of a meaningful name. One coding technique is to continue to use the original (meaningful name) and to use macros to map to a different external name.

```
1  #define comms_inport_1 E1234
2  #define comms_inport_2 E1235
```

This approach suffers from the problem that there are two names associated with every object, not a good state of affairs from a program maintenance point of view. So, care needs to be taken that the alternative macro-derived names are not used directly.

C90 had a six character limit. Such a limit is very low and is an ideal that only a few, ultra-portable programs should still aspire to. However, it is possible that some C90 translators never migrate to the C99 limit (it being uneconomical to upgrade them).

identifier 792 number of characters

The issue of identifier length is discussed more fully elsewhere.

---

footnote 13

13) Implementations should avoid imposing fixed translation limits whenever possible. 284

Commentary

This is only a suggestion to implementors, not a requirement. It implies that implementations ought to use dynamically allocated data structures, rather than fixed-size ones.

Common Implementations

Most implementations dynamically allocate data structures for most constructs. The developer's desire for translators that translate at reasonable rates means that there are trade-offs associated with the use of fixed-size data structures in some areas.

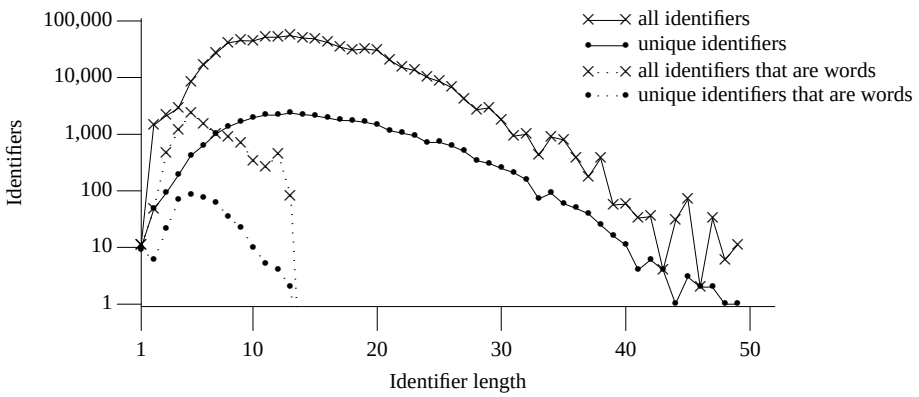


Figure 283.1: Number of identifiers, with external linkage, having a given length. Based on the translated form of this book's benchmark programs. Information on the length of all identifiers in the visible source is given elsewhere (see Figure 792.7).

Coding Guidelines

The developer has no control over the design of an implementation. Although implementations do not go out of their way to make inefficient use of host resources, there is not always the commercial incentive, on some hosts, to improve the quality of a translator.

---

285 — 4095 external identifiers in one translation unit

limit external identifiers

Commentary

This limit may appear to be generous. But, it includes identifiers declared both by the developer and the implementation (when a system header is included). This limit may be reached in automatically generated code. The standard does not define a per program limit. This is mainly because some linkers are not provided by the translator vendor and are in many ways outside of these vendors' control.

C90

511 external identifiers in one translation unit

C++

The following is a non-normative specification.

External identifiers in one translation unit [65536] Annex Bp2

Common Implementations

Most vendors include a large number of identifiers in their system headers. This is particularly true on workstations where the total number of identifiers declared in system headers can exceed 15,000 (see Table 1897.1). Developers have no control over the contents of these headers.

Usage

External declaration usage information is given elsewhere (see Figure 1810.1).

Table 285.1: Number of identifiers with external linkage (total 487), and total number of identifiers (total 810), implementations are required to declare in the standard headers.

Header	External Identifiers	Total Identifiers	Header	External Identifiers	Total Identifiers
<assert.h>	1	2	<signal.h>	2	12
<complex.h>	66	71	<stdarg.h>	3	5
<ctype.h>	15	15	<stdbool.h>	0	4
<errno.h>	1	4	<stddef.h>	0	5
<fenv.h>	11	24	<stdint.h>	0	38
<float.h>	0	31	<stdio.h>	49	65
<inttypes.h>	6	62	<stdlib.h>	36	37
<iso646.h>	0	11	<string.h>	22	24
<limits.h>	0	19	<tgmath.h>	0	60
<locale.h>	2	10	<time.h>	9	15
<math.h>	184	203	<wchar.h>	59	68
<setjmp.h>	2	3	<wctype.h>	18	22

---

286 — 511 identifiers with block scope declared in one block

identifiers number in block scope

Commentary

This limit may be reached in automatically generated code. In human-written code more than 10 identifiers declared in block scope is uncommon.

C90

127 identifiers with block scope declared in one block

C++

The following is a non-normative specification.

Annex Bp2 *Identifiers with block scope declared in one block [1024]*

Common Implementations

Most implementations take advantage of the scoping nature of blocks to create symbol table information when the declaration is encountered and to remove it (freeing up the storage used) when the block scope terminates. For implementations that operate in a single pass, generating machine code on a basic block basis, this can result in considerable storage savings. High-powered optimizing translators may still generate machine code in a single pass, but they usually build a tree representing all of the statements and expressions within each function. This means that information on block scope declarations cannot be freed up at the end of the block in which they occur.

Coding Guidelines

Having a large number of objects defined in the same block may be an indicator that a function definition has grown too large and needs to be split up, or an indicator that a structure type needs to be created. Although this is a design issue, there is a potential impact on comprehension effort. However, your author knows of no method of comparing the comprehension effort required for the various cases and so is silent on the subject.

Usage

The 53,630 function definitions in the translated form of this book's benchmark programs contained: definitions of 76 structure, union or enumeration types that included a tag; 6 **`typedef`** definitions; and definitions of 70 enumeration constants.

---

limit
macro definitions — 4095 macro identifiers simultaneously defined in one preprocessing translation unit 287

Commentary

This limit may appear to be generous. But, it includes macro identifiers declared both by the developer and the implementation (when a system header is included). The standard does not specify limits on the bodies of macro definitions. This is something that usually occupies much more storage than the identifier itself.

This limit may be reached in automatically generated code.

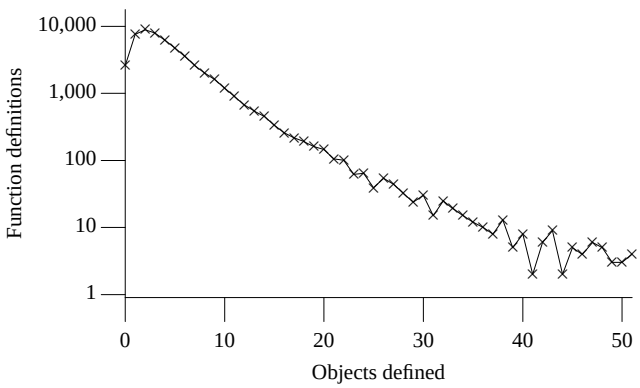


Figure 286.1: Number of function definitions containing a given number of definitions of identifiers as objects. Based on the translated form of this book's benchmark programs.

C90

1024 macro identifiers simultaneously defined in one translation unit

C++

The following is a non-normative specification.

Macro identifiers simultaneously defined in one translation unit [65536] Annex Bp2

Common Implementations

Most vendors include a large number of identifiers in their system headers. This is particularly true on workstations where the total number of identifiers declared in system headers can exceed 15,000 (see Table 1897.1). Developers have no control over the contents of these headers. It would not be uncommon for the total number of macros in a translation unit to exceed this limit (assuming an appropriate number of system headers are included). [98] footnote 3

There are several public domain preprocessors that might be of use if this translator limit on number of macro identifiers is encountered. However, if the problem is caused by lack of storage on the host where the translation is performed, such a tool may not be of practical use. Using a different preprocessor, from the one provided as part of the implementation also introduces the problem of ensuring that any predefined, by one preprocessor, macro names are also defined with the same bodies when another preprocessor is used.

---

288 — 127 parameters in one function definition

limit parameters in definition

Commentary

This limit is rarely reached except in automatically generated code, even then it is rare.

C90

31 parameters in one function definition

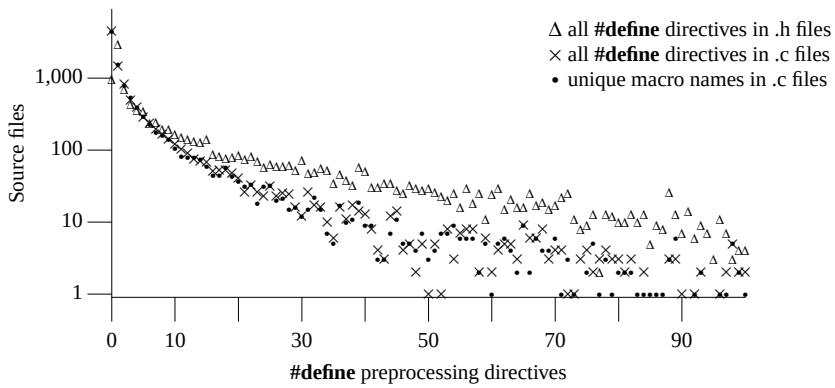


Figure 287.1: Number of source files containing a given number of identifiers defined as macro names in **`#define`** preprocessing directives. Unique macro name counts an identifier once, irrespective of the number of **`#define`** directives it appears in. Based on the visible form of the `.c` and `.h` files.

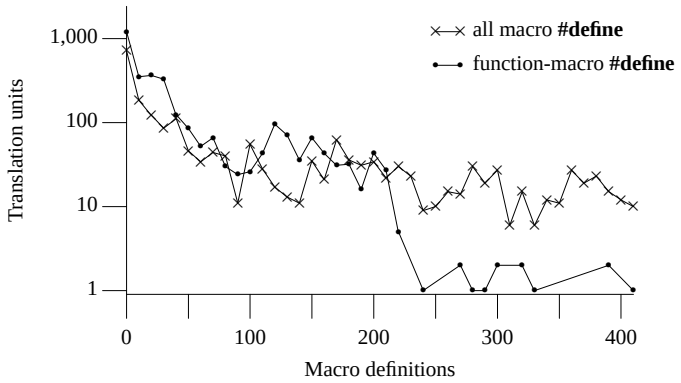


Figure 287.2: Number of translation units containing a given number of evaluations of **#define** preprocessing directives, excluding the contents of system headers, during translation of this book's benchmark programs (there were a total of 1,432,735 macros defined, of which 313,620 were function-like macros).

C++

The following is a non-normative specification.

Annex Bp2 *Parameters in one function definition [256]*

Common Implementations

Few hosted implementations place restrictions on the number of parameters in a function definition. Having one parameter on a stack is much the same as having 100. However, storage-limited execution environments (invariably freestanding) often limit the maximum number of parameters in a function definition.

The C binding for the GKS Standard[643] did manage to exceed the C90 limit, but this is uncommon.

Coding Guidelines

file scope 407 Some coding guideline documents recommend that use of file scope objects be minimized.[1026] This has the consequence of increasing the number of parameters in function definitions. Other guideline documents recommend keeping the number of parameters below a certain limit to reduce the possibility of developers making mistakes (by passing arguments in the incorrect order). Possible alternatives include the following:

- *Relying on file scope objects.* Out of sight, out of mind— developers could easily forget to assign to these objects. Alternatively, once an object has file scope, any number of unexpected functions might also reference it, creating unintended dependencies.
- *Declaring a structure to hold the parameter values.* The arguments now need to be assigned to the members of the structure. The names of these members, if well chosen, could provide a useful reminder of the appropriate value to assign. The disadvantage is that there is no automatic checking when new parameters, in the form of new members, are added, potentially resulting in the new parameters being passed in existing invocations as uninitialized members.
- *Passing as much information as possible through parameters.*

There have been no empirically based studies whose results might be used as the basis for calculating which information-passing method has the optimal cost/benefit.

---

function call number of arguments — 127 arguments in one function call 289

Commentary

limit 288 parameters in definition Functions declared using the ellipsis notation can be called with arguments that exceed this limit, while their definitions do not exceed the limit on the number of parameters.

This limit is rarely reached except in automatically generated code, even then it is rare.

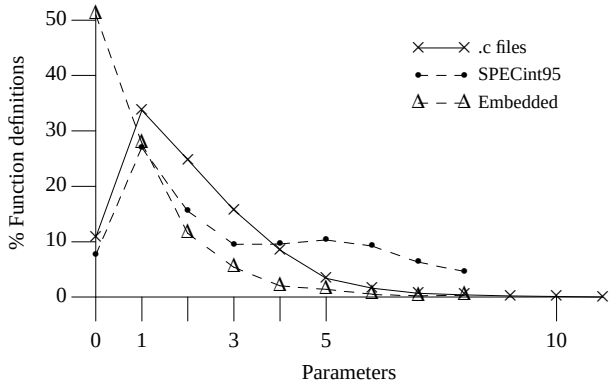


Figure 288.1: Percentage of function definitions appearing in the source of embedded applications (5,597 function definitions), the SPECint95 benchmark (2,713 function definitions), and the translated form of this book's benchmark programs (53,719 function definitions) declared to have a given number of parameters. The embedded and SPECint95 figures are from Engblom.[390]

C90

31 arguments in one function call

C++

The following is a non-normative specification.

Arguments in one function call [256] Annex Bp2

Common Implementations

Few hosted implementations place restrictions on the number of arguments passed in one function call. However, storage-limited execution environments (invariably freestanding) sometimes have limits on the number of bytes available on the function call stack.

290 — 127 parameters in one macro definition limit macro parameters

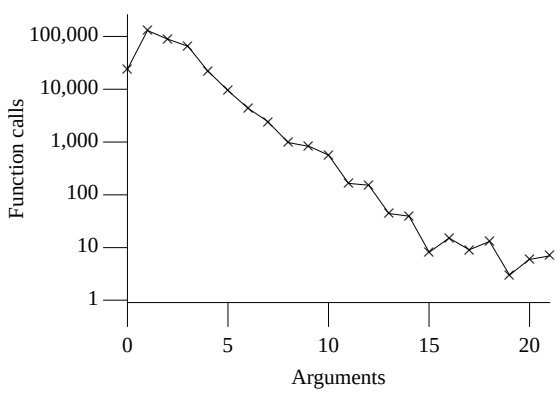


Figure 289.1: Number of function calls containing a given number of arguments. Based on the translated form of this book's benchmark programs.

Commentary

macro 1933 function-like

Function-like macro definitions are sometimes used to provide an alternative to an actual function call. These limits ensure that such definitions can handle at least as many parameters as function definitions.

C90

31 parameters in one macro definition

C++

The following is a non-normative specification.

Annex Bp2

Parameters in one macro definition [256]

Common Implementations

A few implementations used fixed-size data structures for macro definitions. The extent to which these will be increased to support the new C99 limit is not known.

Coding Guidelines

In the case where the macro body is not syntactically a function body, a large number of parameters may be the most reliable method of ensuring that the intended objects are accessed. Because macro bodies are expanded at the point of reference, the objects visible at that point (not the point of definition) are accessed.

---

limit arguments in macro invocation

— 127 arguments in one macro invocation

Commentary

... arguments 1923 macro

It is now possible, in C99, to define macros taking a variable number of arguments, using a similar principle to that used in function definitions. Although the arguments corresponding to the `...` notation are treated as a single parameter, inside the body of the macro definition, not individual ones.

C90

31 arguments in one macro invocation

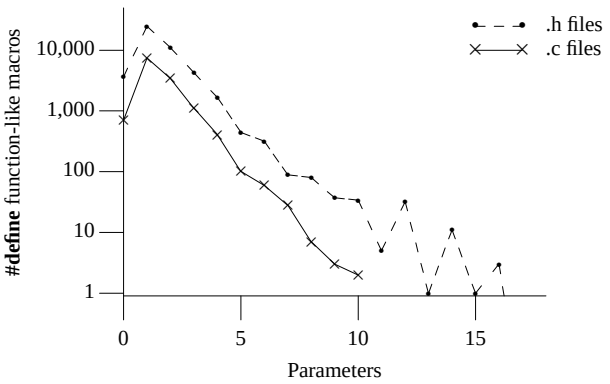


Figure 290.1: Number of function-like macro definitions having a given number of parameter declarations. Based on the visible form of the `.c` and `.h` files.

C++

The following is a non-normative specification.

Arguments in one macro invocation [256] Annex Bp2

Common Implementations

Some implementations limit the size (e.g., the number of characters) of an argument (an early version of Microsoft C[932] had a 256-character limit).

292 — 4095 characters in a logical source line

limit characters on line

Commentary

A logical line is created from a physical line after any line splicing has taken place in translation phase 2. Line splicing is only really needed in macro definitions. This limit can really be thought of as applying to the number of characters in a macro definition.

118 translation phase 2

Note that this limit does not apply to the result of any macro expansion. The C Standard defines a token-based preprocessor; characters and line length need not enter into the macro expansion process.

macro replacement

This limit may, rarely, be reached in automatically generated code.

C90

509 characters in a logical source line

C++

The following is a non-normative specification.

Characters in a logical source line [65536] Annex Bp2

Other Languages

Fortran (prior to Fortran 90) had a limit of 80 characters— the width of an IBM punched card.

Common Implementations

Some implementations use a fixed-length buffer for handling single logical source lines. Others use a fixed-length buffer and handle those situations where the length of a logical line exceeds the length of that buffer as a special case. The environmental limit on the minimum number of characters that may be supported on a physical line may affect translators written in C.

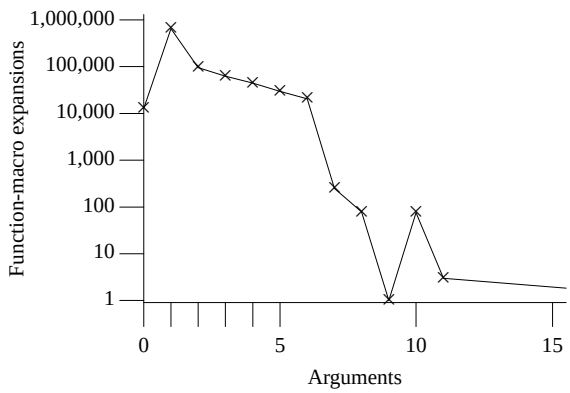


Figure 291.1: Number of function-like macro expansions containing a given number of arguments, excluding expansions that occurred while processing system headers, during translation of this book's benchmark programs.

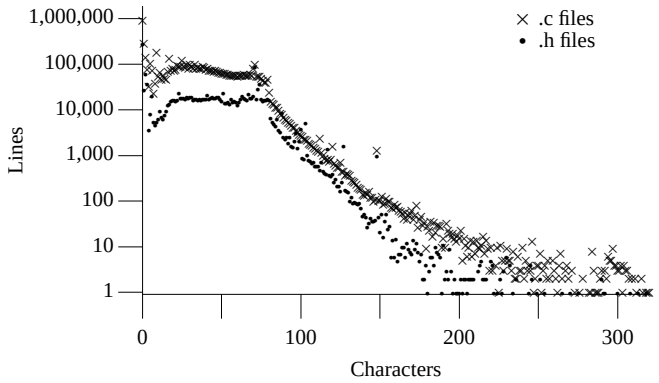


Figure 292.1: Number of physical lines containing a given number of characters. Based on the visible form of the `.c` and `.h` files.

limit string literal

— 4095 characters in a character string literal or wide string literal (after concatenation) 293

Commentary

translation phase 135 6

limit characters on line 292

This limit applies after translation phase 6. If the limit on the number of characters in a logical line is taken into account then, allowing for the delimiting quote characters, the only way of reaching or exceeding this limit without exceeding any other limits is via concatenation. Longer strings, than this limit, can be created by copying character values into object storage. But, these would not be string literals.

C90

509 characters in a character string literal or wide string literal (after concatenation)

C++

The following is a non-normative specification.

Annex Bp2 *Characters in a character string literal or wide string literal (after concatenation) [65536]*

Common Implementations

Some implementations use a fixed-length buffer to hold the characters making up a preprocessing token (in this case a string literal). The characters forming the string literal are rarely held on a linked list. Other implementations use a string-handling package to look after the details of manipulating variable-length string literals and have no internal length restrictions.

Coding Guidelines

If a very long string literal is needed by the application, it makes sense to try to create it as a single entity. String literals containing more than a few hundred characters are rare enough not to be worth a coding guideline.

limit minimum object size

— 65535 bytes in an object (in a hosted environment only) 294

Commentary

Many CISC processors have an efficient 16-bit addressing mode to access objects. It is often possible to create and access objects that exceed this addressing range, but the implementation and execution time overhead can be much higher. In many ways this limit can be seen as giving permission for implementations to stay within the natural addressing structure of their target processor (should it be a 16-bit one).

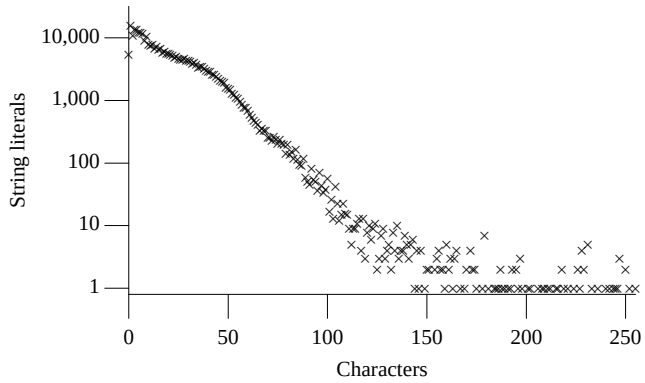


Figure 293.1: Number of character string literals containing a given number of characters (i.e., their length). Based on the visible form of the `.c` files.

The standard does not say anything about the storage duration of objects of this size; does it apply to all of them or at least one of them? There is no specification requiring that it be possible to define more than one of these objects, or for several smaller objects whose total size is 65,535 bytes to be supported.

This limit matches the corresponding minimum limits for `size_t` and `ptrdiff_t`. This limit means that for a translator where the type **`int`** is represented in 16 bits, the typedefs `size_t` and `ptrdiff_t` must have ranks at least equal to the type **`long`**.

Many freestanding environments don't even have 64 K bytes of memory in total. However, the standard does not specify a minimum object size that must be supported in these environments.

Committee Discussion DR #266

Translation limits do not apply to objects whose size is determined at runtime.

C90

32767 bytes in an object (in a hosted environment only)

C++

The following is a non-normative specification.

Size of an object [262144] Annex Bp2

Common Implementations

The maximum size of an object that can be defined may depend on its storage duration. On the basis that most function definitions only use a small amount of local data, some processor designers choose to ignore the relatively rare case of large amounts of local storage being required. The addressing modes needed to access large local objects, with a few instructions, may not be available (instead, relatively long sequences of instructions need to be used). These processor-based limitations can lead to translator vendors deciding not to go to the trouble of supporting very large objects having automatic storage duration.

If support for objects larger than 64 K is needed, it is most likely to be available via allocated storage. However, in some cases there may be limitations caused by host environment restrictions on the amount of storage that can be dynamically allocated in one contiguous storage area.

Linkers sometimes place restrictions on the maximum size of an object that can be statically allocated. This will affect objects with static storage duration.

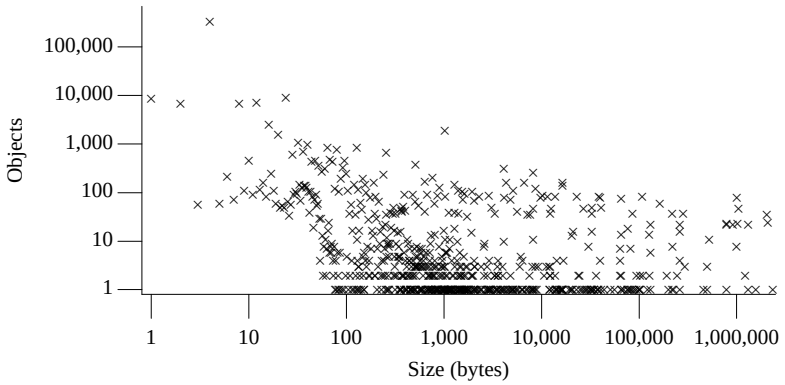


Figure 294.1: Number of objects requiring the specified amount of storage. Based on the translated form of this book's benchmark programs, using integer types whose sizes were: `sizeof(short) == 2`, `sizeof(int) == 4`, and `sizeof(long) == 4`; and alignment requirements that were a multiple of a types size.

Coding Guidelines

Although the standard may require that it be possible to define an object of the specified size, it is silent on the circumstances in which a program containing such a definition must be capable of executing. The standard does not provide any mechanism for verifying that a particular function invocation will successfully start to execute (i.e., not generate a stack overflow). In the case of static storage allocation, the program will either start to execute, or fail to start executing.

Use of dynamic allocation for objects does provide a degree of developer control of the situation where the allocation request fails. The disadvantage of such allocation methods is that it puts more responsibility for getting things right onto the developers' shoulders. Handling execution environment object storage limitations is a design and algorithmic issue that is outside the scope of these coding guidelines.

---

limit #include nesting

— 15 nesting levels for **`#include`**d files 295

Commentary

This limit makes no distinction between system headers and developer-written header files. However, an implementation is required to support its own system headers whose contents are defined by the standard. If a particular implementation chooses to use nested **`#include`**s, then it is responsible for ensuring that these do not prevent a translator from meeting its obligations with regard to this limit.

Use of nested **`#include`**s requires that the source file containing each of the **`#include`** directive be kept open, while the included file is processed. Supporting 15 nesting levels invariably requires keeping at least 17 (the top-level source file and the file holding the generated code also need to be counted) files open simultaneously. Using the `fsetpos` library function to record the current file position, at the point the **`#include`** occurs, and closing the current file before opening the nested include is possible; however, your author has never heard of an implementation that uses it.

C90

8 nesting levels for #included files

C++

The following is a non-normative specification.

Annex Bp2

Nesting levels for #included files [256]

Common Implementations

This limit is not just about data structures within the translator. Most environments have limits on the number of files that a process can have open simultaneously. This limit may be soft in the sense that the developer can modify it, or hard in that the limit is built into the OS (requiring a kernel rebuild to change it).

Coding Guidelines

Developers have no control over the nesting used by system headers. These headers are essentially black boxes, so it is permissible to ignore any nesting that occurs within them, from the point of view of calculating the maximum **#include** nesting level.

Does the depth of nesting of **#include** files affect the cost of ownership of source code?

Headers that only contain information specific to a particular application area, or even data type, would seem to be following the principles of information-hiding. The nesting of headers might be mapped to the corresponding nesting of application data structures. A practical problem associated with information-hiding in headers is locating the header that contains a particular identifier declaration. Program development environments rarely provide tools for handling headers in a structured fashion.

In a traditional development environment the source code editor does not usually have any knowledge of the character sequences it is displaying, although some editors do support a tags facility (enabling a database of identifier tags and the files that reference them to be built). Syntax highlighting is also becoming common and C++ style class browsers are growing in popularity, but structured support for displaying header file contents is still uncommon.

Given the support tools that are likely to be available to a developer, a limit on the nesting of **#include** directives could provide a benefit by reducing developer effort when browsing the source of various header files. However, limiting the nesting to which **#include** directives can occur could increase configuration-management costs, or result in poorly structured source files. The cost/benefit issues are complex and these coding guidelines say nothing more on the issue.

---

296 — 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)

limit
case labels

Commentary

The limit chosen in C90 had a rationale behind it. The value used for C99 has been increased in line with the percentage increases seen in other limits.

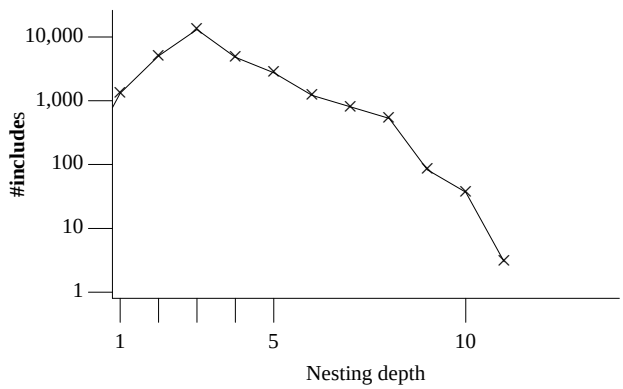


Figure 295.1: Number of **#include** preprocessor directives, that contain the quote-delimited form of header name (occurrences of the < > delimited form were not counted), having a given nesting depth. Based on the translated form of this book's benchmark programs.

C90

*257 case labels for a **switch** statement (excluding those for any nested **switch** statements)*

switch statement 1748 The intent here was to support **switch** statements that included 256 unsigned character values plus EOF (usually implemented as -1).

C++

The following is a non-normative specification.

Annex Bp2 *Case labels for a **switch** statement (excluding those for any nested **switch** statements) [16384]*

Common Implementations

selection statement syntax 1739 The number of **case** labels in a **switch** statement may affect the generated code. For instance, a processor instruction designed to indirectly jump based on an index into a jump table may have a 256 (bytes or addresses) limit on the size of jump table supported.

Coding Guidelines

The number of **case** labels in a **switch** statement is an indication of the complexity of the application, or the algorithm. Splitting a **switch** statement into two or more **switch** statements solely for the purpose of reducing the number of **case** labels within an individual **switch** statement has costs and benefits. The following are some of the issues:

- The size of a large **switch** statement, in the visible source, can affect the ease of navigation of the function containing it.
- In some environments (e.g., freestanding environments) there may be resource limitations (these may occur as a result of processor architecture or internal limits of the translator used); for instance, generating to case tables when there are large ranges of unused case values may make inefficient use of storage, or mapping to if/else pairs may cause a critical section of code to execute too slowly.
- The code complexity is increased (e.g., a conditional test that did not previously exist needs to be introduced).
- There may not be semantically meaningful disjoint sets of **case** labels that can form a natural division into separate **switch** statements.

---

limit members in struct/union — 1023 members in a single structure or union 297

Commentary

This limit does not include the members of structure tags, or typedef names, that have been defined elsewhere and are referenced in a structure or union definition.

This limit may be reached in automatically generated code.

C90

127 members in a single structure or union

C++

The following is a non-normative specification.

Annex Bp2

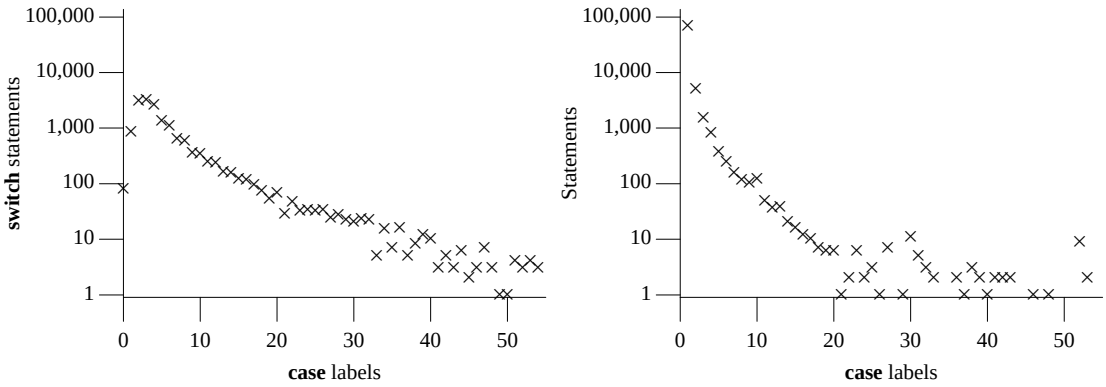


Figure 296.1: Number of **switch** statements containing the given number of **case** labels (left) and number of individual statements labeled by a given number of **case** labels (right). Based on the visible form of the `.c` files. Note that counts do not include occurrences of the **default** label.

Data members in a single class, structure or union [16384]

Coding Guidelines

The organization of the contents of data structures is generally determined by the application and algorithms used. While it may be difficult to imagine a structure definition containing a large number of members without some subset sharing a common characteristic, which enables them to be split into separate definitions; this does not mean that structures with large numbers of members cannot occur for a good reason. This issue is discussed in more detail elsewhere.

299 limit struct/union nesting

It is unlikely that a, human-written, union definition would ever contain a large number of members.

Usage

Measurements of classes,[1455] in large Java programs, have found that the number of members follows the same pattern as that in C (see Figure 297.1).

---

298 — 1023 enumeration constants in a single enumeration

limit enumeration constants

Commentary

This C99 limit has a value comparable to the increased limits of other constructs sharing the same C90 limit.

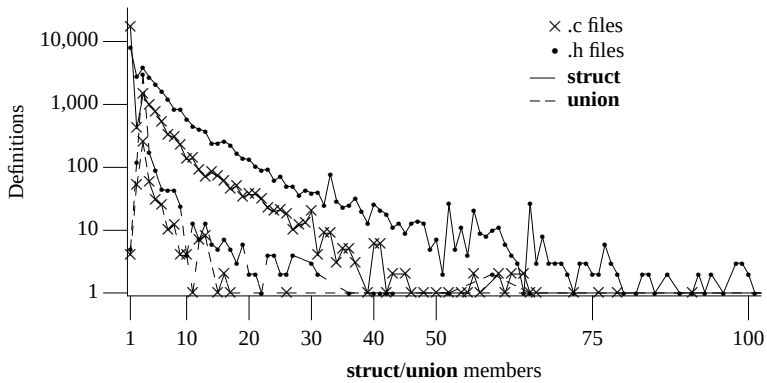


Figure 297.1: Number of structure and union type definitions containing the given number of members (members in any nested definitions are not included in the count of members of the outer definition). Based on the visible form of the `.c` and `.h` files.

C90

127 enumeration constants in a single enumeration

C++

The following is a non-normative specification.

Annex Bp2 *Enumeration constants in a single enumeration [4096]*

Common Implementations

The only known implementation restrictions on enumeration constants in an enumeration is the amount of available memory.

Coding Guidelines

Enumerators are unstructured in the sense that it is not possible to break down an enumerator into component parts to be included in some higher-level enumerator.

Would an application ever demand a large number of enumeration constants in a single enumerator? There are some cases where a large number do occur; for instance, specifying the tokens in the SQL/2 grammar requires 318 enumeration constants. Limiting the number of enumeration constants in an enumeration would not appear to offer any benefits, especially since there is no mechanism (like there is for structure members) for creating hierarchies of enumeration constants.

---

limit struct/union nesting — 63 levels of nested structure or union definitions in a single struct-declaration-list

Commentary

The specification of definitions, rather than types, suggests that this limit does not include nested structures and unions that occur via the use of typedef names.

C90

15 levels of nested structure or union definitions in a single struct-declaration-list

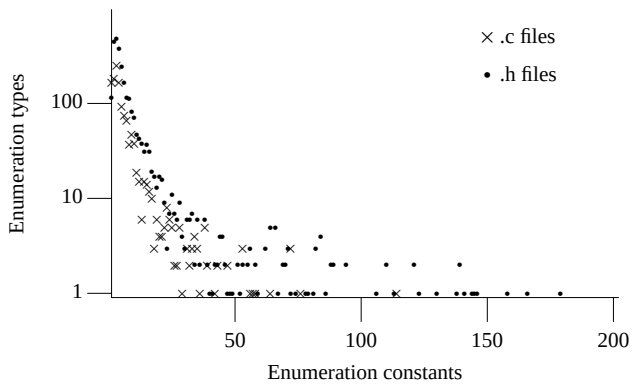


Figure 298.1: Number of enumeration types containing a given number of enumeration constants. Based on the visible form of the `.c` and `.h` files (also see Figure 1439.1).

C++

The following is a non-normative specification.

Levels of nested class, structure, or union definitions in a single `struct-declaration-list` *[256]* Annex Bp2

Other Languages

Fortran 90 and Common Lisp do not support the lexical nesting of record declarations.

Common Implementations

Nesting of definitions is part of the language syntax and usually implemented using a table-driven syntax analyzer. Most nested structure and union definitions occur at file scope; that is, they are not nested within other constructs. However, even if a declaration occurs in block scope, it is likely that any internal table limits will not be exceeded by a definition nested to 63 levels. This limit is only half that of nested block levels, even if creating a new level of structure nesting consumes 2 to 3 times as many table entries as a new level of nested block, there is likely to be sufficient table entries remaining to handle a deeply nested structure definition.

Coding Guidelines

Textual nesting of structure and union definitions is not necessary. C contains a mechanism (typedef names or tags) that removes the need for any textual nesting within structure or union definitions. Is there some optimal level of nesting, or can developers simply create whatever declarations suit them at the time?

The following are some of the benefits of using textually nested definitions:

- Having a definition nested within the structure of which it is a subcomponent highlights, to readers, the close association between the two types.
- Nested definitions may reduce reader effort (e.g., source code scanning) in locating a needed definition (e.g., the type of a member will be visible in the source next to where the member appears).
- If a single instance of a definition is needed, there is no need to create a tag name for it.

Some of the costs of using textually nested definitions include:

- Given the typical source code indentation strategies used for nested definitions, it is likely that deeply nested definitions will cause the same layout problems as deeply nested blocks (this issue is discussed elsewhere). 277 limit block nesting
- There is a reader expectation that a reference to a tag name, either refers to a definition nested within the current definition or that is not nested within any other definition. Such an expectation increases search costs (because the search is performed visually rather than via, say, an editor search command).
- C++ compatibility— a structure/union definition is treated as a scope in C++, which means that names defined within it are not visible outside of it. Any references to tags nested within other definitions will cause a C++ translator to issue a diagnostic and the definition will have to be unnested before these references will be acceptable to a C++ translator. 402 scope kinds of

There are also costs and benefits associated with nested definitions, whether such definitions are created through textual occurrence in the source or the use of tag or typedef names. The alternative to using nested definitions is to have a single level of definition (i.e., all structure or union members have a scalar or array type).

The issues involved in deciding the extent to which members having a shared semantic association should be contained in a single structure definition (i.e., used whenever that member is required, sometimes creating a nested structure) or duplicated in more than one structure definition (which does not require a nested structure to be created) include:

member selection 1031

- References (e.g., as an operand in an expression) to members nested within other definitions requires a sequence of member-selection operators (it may be possible to visually hide these behind a macro invocation). The visible source needed for accessing deeply nested members may impact the layout of the containing expression (e.g., a line break may be needed).
- As source code evolves, the information represented by a member that was once unique to one structure definition may need to be represented in other structure definitions. Creating a new type containing the related members may have many benefits (moving a member to such a definition changes what was a nonnested member into a nested member). However, the cost of adding a nesting level to an existing type definition can be high. The editing effort will be proportional to the number of occurrences of the moved members in the existing code, which requires the appropriate additional member-selection operation to be added.

categorization 0

- Type definitions are not always based on the categorization attributes of the application or algorithm. How a source code manipulates that information also needs to be considered. For instance, if some function needs to access particular information and takes as an argument a structure object that might reasonably contain a member holding that information, it might be decided to define the member holding that information in that structure type rather than another one.

developers organized knowledge 0

- The advantages of organizing information according to shared attributes is discussed in the introduction.

At the time of this writing algorithmic methods for optimally, or even approximately optimal, selecting structure type hierarchies are not known. Some of the considerations such an algorithm would need to take into account: the effort needed by readers to recall which structure type included which member, the effort needed to modify the types as the source evolved over time, and the structuring requirements caused by the need to pass arguments or create pointers to members.

Example

```
1  struct R {
2            struct {
3                   int mem_1;
4                   } mem_2;
5            };
6  struct S {
7            /*
8             * A tag is only needed if the structure type is referred to.
9             */
10           struct T {
11                    int mem_1;
12                    } mem_2;
13           };
14
15 struct U {
16           int mem_1;
17           };
18 struct V {
19           struct U mem_2;
20           };
```

5.2.4.2 Numerical limits

numerical limits

An implementation is required to document all the limits specified in this subclause, which are specified in the headers **<limits.h>** and **<float.h>**. 300

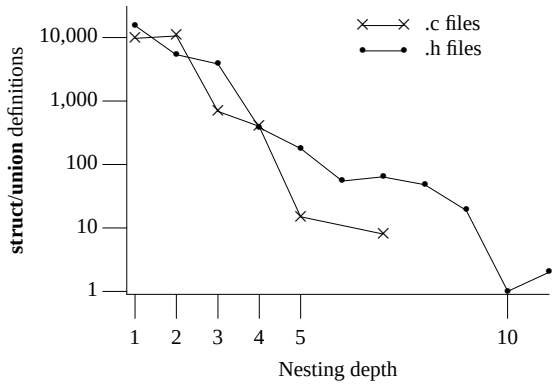


Figure 299.1: Number of structure and union type definitions containing the given number nested members that are textually structure and union type definitions (i.e., definitions using **{ }** not typedef names). Based on the visible form of the `.c` and `.h` files.

Commentary

Telling developers to look at the contents of the headers **<limits.h>** and **<float.h>** could well suffice as documentation.

C90

A conforming implementation shall document all the limits specified in this subclause, which are specified in the headers ***<limits.h>*** *and* ***<float.h>****.*

C++

Header ***<climits>*** *(Table 16): . . . The contents are the same as the Standard C library header* ***<limits.h>****.* 18.2.2p2

Header ***<cfloat>*** *(Table 17): . . . The contents are the same as the Standard C library header* ***<float.h>****.* 18.2.2p4

Other Languages

Many languages provide no means of obtaining such information about an implementation. The size of all scalar data types is predefined in Java. There are no implementation decisions to be made. The Java classes—`java.lang.Character`, `java.lang.Integer` and `java.lang.Long`— contain members giving minimum and maximum values of those types. There is no such class for the type **short**.

Common Implementations

There are two main kinds of implementations. Those in which **int** is 16 bits and those in which **int** is 32 bits. A third kind might be added, those in which **int** is not one of these two values (24 and 40 have been used). Processors supporting 128-bit integer types, where **int** might naturally be 64 bits, are beginning to appear.[27]

---

301 Additional limits are specified in **<stdint.h>**.

Commentary

This header contains declarations of integer types having specified widths, along with corresponding limits macros.

C90

Support for these limits and the header that contains them is new in C99.

C++

Support for these limits and the header that contains them is new in C99 and is not available in C++.

Forward references: integer types <stdint.h> (7.18). 302

5.2.4.2.1 Sizes of integer types <limits.h>

---

integer types sizes

303 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives.

Commentary

macro object-like 1931 In other words they represent object-like macros. The difference between this statement and one requiring that the values be integral constant expressions is that the **sizeof** and cast operators cannot appear in a **#if** preprocessing directive (or to be more exact, a sequence of preprocessing tokens in this context is not converted to tokens and **sizeof** or type names are not treated as such).

These macros were created to give names to values that developers are invariably aware of, may make use of, and for which they may have defined their own macros had the standard not provided any.

C++

17.4.4.2p2 *All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in* ***#if*** *preprocessing directives, unless explicitly stated otherwise.*

18.2.2p2 *Header* ***<climits>*** *(Table 16): . . . The contents are the same as the Standard C library header* ***<limits.h>***

Other Languages

The equivalent identifiers in Java contain constant values. Ada and Fortran (90) use a function notation for returning representation properties of types and objects.

Common Implementations

These values can all be implemented using strictly conforming C constructs (although the numerical values used may vary between implementations). That is not to say that all vendors' implementations do it in this way. For some of these macros it is possible to use the same definition in all environments (without affecting the existing conformance status of programs). However, this usage is not very informative:

```
#define UINT_MAX (-1U)
#define ULONG_MAX (-1UL)
#define ULLONG_MAX (-1ULL)
```

The definition of the macros for the signed types always depends on implementation-defined characteristics. For instance,

```
#define INT_MAX 32767        /* 16 bits */
#define INT_MIN (-32767-1)
```

Coding Guidelines

The standard specifies values, not the form the expression returning those values takes. For instance, the replacement for INT_MIN is not always the sequence of digits denoting the actual value returned.

317 int minimum value

Table 303.1: Number of identifiers defined as macros in **<limits.h>** (see Table 770.3 for information on the number of identifiers appearing in the source) appearing in the visible form of the .c and .h files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
LONG_MAX	47	28	CHAR_MAX	15	8	CHAR_BIT	36	3
INT_MAX	106	17	INT_MIN	17	7	SCHAR_MIN	12	2
UINT_MAX	30	14	UCHAR_MAX	16	5	LLONG_MAX	0	1
SHRT_MAX	20	13	CHAR_MIN	9	5	ULLONG_MAX	0	0
SHRT_MIN	19	12	SCHAR_MAX	13	4	LLONG_MIN	0	0
USHRT_MAX	12	11	MB_LEN_MAX	15	4			
ULONG_MAX	85	10	LONG_MIN	23	3			

304 Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

*_MAX same type as *_MIN same type as

Commentary

CHAR_BIT and MB_LEN_MAX have no implied integer type. This requirement maintains the implicit assumption that use of one of these identifiers should not cause any surprises to a developer. There could be some surprises if promotions occurred; for instance, if the constants had type **unsigned long long**. There is no literal suffix to indicate a character or **short** type, so the type can only be the promoted type.

Common Implementations

Suffixes are generally used, rather than hexadecimal notation, to specify unsigned types.

305 Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

Commentary

The C Standard does not specify the number of bits in a type. It specifies the minimum and maximum values that an object of that type can represent. An implementation is at liberty to exceed the limits specified here. It cannot fail to meet them. Except for the character types, a type may also contain more bits in its object representation than in its value representation.

494 integer types relative ranges
574 object representation
595 value representation

Common Implementations

The values that are most often greater than the ones shown next are those that apply to the type **int**. On hosted implementations they are often the same as the corresponding values for the type **long**. On a freestanding implementation the processors' efficiency issues usually dictate the use of smaller numeric ranges, so the minimum values shown here are usually used. The values used for the corresponding character, **short**, **long**, and **long long** types are usually the same as the ones given in the standard.

The Unisys A Series[1390] is unusual in not only using sign magnitude, but having a single size (six bytes) for all non-character integer types (the type **long long** is not yet supported by this vendor's implementation).

```
#define SHRT_MIN        (-549755813887)
#define SHRT_MAX          549755813887
#define USHRT_MAX         549755813887U
#define INT_MIN         (-549755813887)
#define INT_MAX           549755813887
#define UINT_MAX          549755813887U
#define LONG_MIN        (-549755813887L)
```

```
#define LONG_MAX          549755813887L
#define ULONG_MAX         549755813887UL
```

The character type use two's complement notation and occupies a single byte.

The C compiler for the Unisys e-@ction Application Development Solutions (formerly known as the Universal Compiling System, UCS)[1391] has 9-bit character types— 18-bit **short**, 36-bit **int** and **long**, and 72-bit **long long**.

Coding Guidelines

Some applications may require that implementations support a greater range of values than the minimum requirements specified by the standard. It is the developer's responsibility to select an implementation that meets the application requirements in this area.

symbolic name 822

These, and other, macro names defined by the standard provide symbolic names for the quantities they represent. The contexts in which these identifiers (usually macros) might be expected to occur (e.g., as operands of certain operators) are discussed in subsections associated with the C sentence that specifies them.

Example

Whether an occurrence of a symbolic name is making use of representation information is not always clear-cut.

```
#include <limits.h>

extern _Bool overflow;
extern int total;

void f(int next_digit_val)
{
/*
 * Check that the calculation will probably not overflow.  Perform
 * arithmetic on INT_MAX, but use a property (being a factor of
 * ten-less-than), not any specific numeric value.  The symbolic
 * name  is still being treated as the maximum representable int value.
 */
if (total < (INT_MAX / 10))
   total = (total * 10) + next_digit_val;
else
   overflow = 1;
}

unsigned int g(unsigned long some_val)
{
/*
 * While the internal representation of the value of UINT_MAX
 * is required to have all bits set, this might be considered
 * incidental to the property it is defined to represent.
 */
return (unsigned int)(some_val & UINT_MAX);
}

void h(int *characteristic)
{
/*
 * Comparing symbolic names against individual literal values is often
 * an indication that use is being made of representation details.  If
 * the comparisons occur together, as a set, it might be claimed that an
 * alternative representation of the characteristics of an implementation
 * are being deduced.  The usage below falls into the trap of overlooking
 * an undefined behavior (unless a translator has certain properties).
 */
switch (INT_MIN)
```

```
41      {
42      case -32767: *characteristic=1;
43                   break;
44
45      case -32768: *characteristic=2;
46                   break;
47
48      case -2147483647: *characteristic=3;
49                   break;
50
51      default: *characteristic=4;
52               break;
53      }
54  }
```

306 14) See "future language directions" (6.11.3). footnote 14

307 — number of bits for smallest object that is not a bit-field (byte) CHAR_BIT macro

CHAR_BIT 8

Commentary

C is not wedded to an 8-bit byte, although this value is implicit in a large percentage of source written in it.

Common Implementations

On most implementations a byte occupies 8 bits. The POSIX Standard requires that CHAR_BIT have a value of 8. The Digital DEC 10 and Honeywell/Multics[179] used a 36-bit word with the underlying storage organization based on 9-bit bytes. Some DSP chips have a 16- or 32-bit character type (this often has more to do with addressability issues than character set sizes).

Coding Guidelines

A value of 8, for this quantity, is welded into many developers' mind-set. Is a guideline worthwhile, just to handle those cases where it does not hold? Developers porting programs from an environment where CHAR_BIT is 8, to an environment where it is not 8 are likely to uncover many algorithmic dependencies. In many application domains the cost of ensuring that programs don't have any design, or algorithmic, dependencies on the number of bits in a byte is likely to be less than the benefit (reduced costs should a port to such an environment be undertaken).

A literal appearing within source code conveys no information on what it represents. If the number of bits in a byte is part of a calculation, use of a name, which has been defined, by the implementation to represent that quantity (even if it is not expected that the program will ever be translated in an environment where the value differs from 8) provides immediate semantic information to the developer reading the source (saving the effort of deducing it).

The CHAR_BIT is both a numeric quantity and represents bit-level information.

Example

In the first function assume a byte contains 8 bits. The second function has replaced the literal that explicitly contains this information, but not the literal that implicitly contains it. The third function contains an implicit assumption about the undefined behavior that occurs if `sizeof(int) == sizeof(char)`.

```
1  #include <limits.h>
2
3  unsigned char second_byte_1(unsigned int valu)
4  {
5  return ((valu >> 8) & 0xff);
```

```
6   }
7
8   unsigned char second_byte_2(unsigned int valu)
9   {
10  return ((valu >> CHAR_BIT) & 0xff);
11  }
12
13  unsigned char second_byte_3(unsigned int valu)
14  {
15  return ((valu >> CHAR_BIT) & ((0x01 << CHAR_BIT) - 1));
16  }
```

SCHAR_MIN value — minimum value for an object of type **signed char** 308

SCHAR_MIN -127 // $-(2^7-1)$

Commentary

The minimum value an 8-bit, one's complement or sign magnitude representation can support.

Common Implementations

A value of -128 is invariably used in a two's complement representation.

Coding Guidelines

two's complement minimum value

Because almost all implementations use two's complement, this value is often assumed to be -128. Porting to processors that use a different integer representation is likely to be very rare event. Designing code to handle both cases increases cost and complexity (leading to faults) for a future benefit that is unlikely to be cashed in. Ignoring the additional value available in two's complement implementations, on the grounds of symmetry or strict standard's conformance, can lead to complications if this value can ever be created (e.g., by a cast from a value having greater rank).

— maximum value for an object of type **signed char** 309

SCHAR_MAX +127 // 2^7-1

UCHAR_MAX — maximum value for an object of type **unsigned char** 310

UCHAR_MAX 255 // 2^8-1

Commentary

Because there are two representations of zero in one's complement and signed magnitude the relation `(SCHAR_MAX-SCHAR_MIN) < UCHAR_MAX` is true for all implementations targeting such environments.

In the case of two's complement the equality `(SCHAR_MAX-SCHAR_MIN) == UCHAR_MAX` is true, provided there are no padding bits in the representation of the type **signed char** (padding bits are never permitted in the representation of the type **unsigned char**, and no implementations known to your author use padding bits in the type **signed char**).

unsigned char 571 pure binary

unsigned char 571 pure binary 316 USHRT_MAX UINT_MAX 319 byte 53 addressable unit

There is a requirement that `UCHAR_MAX` equals $2^{\texttt{CHAR\_BIT}} - 1$. Equivalent requirements do not hold for `USHRT_MAX` or `UINT_MAX`

A byte is commonly thought of as taking on the range of values between zero and `UCHAR_MAX` and occupying `CHAR_BIT` bits. In other words it is treated as an unsigned type.

Other Languages

The Java class `java.lang.Character` contains the member:

```
1   public static final char MAX_VALUE = '\uffff';
```

Coding Guidelines

The macro `UCHAR_MAX` has a value that has an arithmetic property. It also has a bitwise property— all bits set to one. The property of all bits set to one is often seen in code that performs bit manipulation (the issue of replacing numeric literals with meaningful identifiers is dealt with elsewhere).

all bits one

822 symbolic name

---

311 — minimum value for an object of type **char**

char minimum value CHAR_MIN

CHAR_MIN *see below*

Commentary

The variability in the value of this macro follows the variability of the choice of representation for the type **char**. It depends on whether **char** is treated as a signed or unsigned type. There is an equivalent set of macros for wide characters.

326 char if treated as signed integer

Other Languages

`java.lang.Character` contains the member:

```
1   public static final char MIN_VALUE = '\u0000';
```

Common Implementations

The value used is often translation-time selectable, via a conditional inclusion in the **<limits.h>** header. The translator using a developer-accessible option to control the choice of type.

---

312 — maximum value for an object of type **char**

CHAR_MAX

CHAR_MAX *see below*

Commentary

The commentary on `CHAR_MIN` is applicable here.

311 char minimum value

Other Languages

`java.lang.Character` contains the member:

```
1   public static final char MAX_VALUE = '\uffff';
```

---

313 — maximum number of bytes in a multibyte character, for any supported locale

MB_LEN_MAX

MB_LEN_MAX 1

Commentary

For the value of this macro to be a translation-time constant, it has to refer to the locales known to the translator. The value must be large enough to support the sequence of bytes needed to specify a change from any supported shift state, to any other supported shift state, for any character available in the later shift state. There is no requirement on the implementation to support redundant changes of shift state.

2017 footnote 152

Common Implementations

Many US- and European-based vendors use a value of 1. The GNU library uses a value of 16.

Coding Guidelines

Using this identifier as the size in the definition, of an array used to hold the bytes of a multibyte character, implies that the bytes stored will not contain any redundant shift sequences.

---

short minimum value — minimum value for an object of type **short int** 314

```
SHRT_MIN -32767 // -(2^15-1)
```

Commentary

The minimum value a 16-bit, one's complement or sign magnitude representation can support.

Other Languages

Java defines **short** as having the full range of a two's complement 16-bit value.

Common Implementations

Most implementations support a 16-bit **short** type.

The token - is a unary operator; it is not part of the integer constant token. The expression -32768 has type **long** (assuming a 16-bit **int** type), ((-32767)-1) has type **int** (assuming that two's complement notation is being used). A value of (-32767-1) is invariably used in a 16-bit two's complement representation.

— maximum value for an object of type **short int** 315

```
SHRT_MAX +32767 // 2^15-1
```

Commentary

short 314 minimum value The commentary on SHRT_MIN is applicable here.

USHRT_MAX — maximum value for an object of type **unsigned short int** 316

```
USHRT_MAX 65535 // 2^16-1
```

Commentary

object representation 574 There is no requirement that USHRT_MAX equals $2^{\texttt{sizeof(short)} \times CHAR\_BIT} - 1$.

Common Implementations

The standard does not prohibit an implementation giving USHRT_MAX the same value as SHRT_MAX (e.g., Unisys A Series[1390]). In practice the additional bit used to represent sign information, in a signed type, is invariably used to represent a greater range of positive values for unsigned types.

---

int minimum value — minimum value for an object of type **int** 317

```
INT_MIN -32767 // -(2^15-1)
```

Commentary

The minimum value a 16-bit, one's complement or sign magnitude representation can support.

Other Languages

Java defines **int** as holding the full range of a 32-bit two's complement value. java.lang.Integer contains the member:

```
1  public static final int MIN_VALUE = 0x80000000;
```

Common Implementations

On most implementations this macro is the same as either SHRT_MIN or LONG_MIN. A value of (-32767-1) is invariably used in a 16-bit two's complement representation, while a value of (-2147483647-1) is invariably used in a 32-bit two's complement representation. 314 short minimum value

A number of DSP processors use 24 bits to represent the type **int**.[970]

Coding Guidelines

Developers often make some fundamental assumptions about the representation of **int**. These assumptions may be true in the environment in which they spend most of their time developing code, but in general are always true. Coding to the minimum limit specified in the standard in a 32-bit environment can be very restrictive and costly. If a program is never going to be ported to a non-32-bit host the investment cost of writing code to the minimum requirements, guaranteed by the standard, may never reap a benefit. This is a decision that is outside the scope of these coding guidelines.

---

318 — maximum value for an object of type **int** INT_MAX

```
INT_MAX +32767 // 2^15-1
```

Other Languages

java.lang.Integer contains the member:

```
1   public static final int MAX_VALUE = 0x7fffffff;
```

Common Implementations

On most implementations this macro is the same as either SHRT_MAX or LONG_MAX.

---

319 — maximum value for an object of type **unsigned int** UINT_MAX

```
UINT_MAX 65535 // 2^16-1
```

Commentary

There is no requirement that UINT_MAX equal $2^{\texttt{sizeof(int)} \times CHAR\_BIT} - 1$. 574 object representation

Common Implementations

The standard does not prohibit an implementation giving UINT_MAX the same value as INT_MAX, which would entail increasing the value of INT_MAX (e.g., Unisys A Series[1390]). In practice the bit used to represent sign information, in a signed type, is invariably included in the value representation for unsigned types (enabling a greater range of positive values to be represented). On most implementations this macro is the same as either USHRT_MAX or ULONG_MAX.

Coding Guidelines

The term *word* has commonly been used in the past to refer to a unit of storage that is larger than a byte. The unit of storage chosen is usually the *natural* size for a processor, much like the suggestion for the size of an **int** object. Like a byte, a word is usually considered to be unsigned, taking on the range of values between zero and UINT_MAX. word range of values 485 int natural size

Like the UCHAR_MAX macro the UINT_MAX macro is sometimes treated as having the bitwise property of all bits set to one. 310 UCHAR_MAX 310 all bits one

---

320 — minimum value for an object of type **long int**

```
LONG_MIN -2147483647 // -(2^31-1)
```

Commentary

The minimum value a 32-bit, one's complement or sign magnitude representation can support.

Other Languages

`java.lang.Long` contains the member:

```
public static final long MIN_VALUE = 0x8000000000000000L;
```

Common Implementations

A value of `(-2147483647-1)` is invariably used in a 32-bit two's complement representation. Prior to the introduction of the type **`long long`** in C99, some implementations targeting 64-bit processors chose to use the value `(-9223372036854775807-1)`, while others extended the language to support a **`long long`** type and kept long at 32 bits.

The number of bits used to represent the type **`long`** is not always the same as, or an integer multiple of, the number of bits in the type **`int`**. The ability to represent a greater range of values (than is possible in the type **`int`**) may be required, but processor costs may also be a consideration. For instance, the Texas Instruments TMS320C6000,[1342] a DSP processor, uses 32 bits to represent the type **`int`** and 40 bits to represent the type **`long`** (this choice is not uncommon). Those processors (usually DSP) that use 24 bits to represent the type **`int`**,[970] often use 48 bits to represent the type **`long`**. The use of 24/48 bit integer type representations can be driven by application requirements where a 32/64-bit integer type representation are not cost effective.[962]

Coding Guidelines

Some developers assume that `LONG_MIN` is the most negative value that a program can generate. The introduction of **`long long`**, in C99, means that this assumption is no longer true.

widest type assumption 494

— maximum value for an object of type **`long int`** 321

`LONG_MAX +2147483647` `// 2^31-1`

Other Languages

`java.lang.Long` contains the member:

```
public static final long MAX_VALUE = 0x7fffffffffffffffL;
```

Common Implementations

The value given in the standard is the one almost always used by implementations.

— maximum value for an object of type **`unsigned long int`** 322

`ULONG_MAX 4294967295` `// 2^32-1`

Commentary

There is no requirement that `ULONG_MAX` equal $2^{\texttt{sizeof(long)} \times CHAR\_BIT} - 1$.

object representation 574

Common Implementations

The standard does not prohibit an implementation giving `ULONG_MAX` the same value as `LONG_MAX` (Unisys A Series[1390]). In practice the additional bit used to represent sign information, in a signed type, is invariably used to represent a greater range of positive values for unsigned types.

The value given in the standard is the one almost always used by implementations.

Coding Guidelines

Some developers assume that `ULONG_MAX` is the largest value that a program can generate. The introduction of the type **`long long`** means that this assumption is no longer true.

widest type assumption 494

323 — minimum value for an object of type **long long int**

LLONG_MIN -9223372036854775807 // $-(2^{63}-1)$

Commentary

The minimum value a 64-bit, one's complement or sign magnitude representation can support. This value is calculated in a way that is consistent with the other _MIN values. However, hardware support for 64-bit integer arithmetic is usually only available on modern hosts, all of which use two's complement (to the best of your author's knowledge).

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

Other Languages

In Java the type **long** uses 64 bits; a **long long** type is not needed (unless support for 128-bit integers were added to the language).

Common Implementations

This macro was supported in those C90 implementations that had added support for **long long** as an extension.

Coding Guidelines

The **long long** data type is new in C99. It will be some time before it is widely supported. A source file that accesses this identifier will fail to translate with a C90 implementation. The issue encountered is likely to be one of nontranslation, not one of different behavior. Deciding to create programs that require at least a 64-bit data type is an application-based issue that is outside the scope of these coding guidelines.

324 — maximum value for an object of type **long long int**

LLONG_MAX +9223372036854775807 // $2^{63}-1$

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

325 — maximum value for an object of type **unsigned long long int**

ULLONG_MAX 18446744073709551615 // $2^{64}-1$

Commentary

There is no requirement that ULLONG_MAX equals $2^{\texttt{sizeof(long long)} \times CHAR\_BIT} - 1$. 574 object representation

C90

Support for the type **unsigned long long** and its associated macros is new in C99.

C++

The type **unsigned long long** is not available in C++.

326 If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**.

char if treated as signed integer

Commentary

integer promotions 675

In most implementations the value of objects of type **char** will promote to the type **signed int**. So the values of objects of type **char** are treated as a **signed integer**. It is only when an object of type **char** is treated as an lvalue, that its signedness becomes a consideration. For instance, on being assigned to, is modulo arithmetic or undefined behavior used for out-of-range values.

lvalue 721

Common Implementations

Most implementations provide a translation-time option that enables a developer to select how the type **char** is to be handled. This option usually causes the translator to internally predefine an object-like macro. The existence of this macro definition is tested inside the **<limits.h>** header to select between the two possible values of the CHAR_MIN and CHAR_MAX macros.

Coding Guidelines

A comparison of the value of CHAR_MIN against SCHAR_MIN (or CHAR_MAX against SCHAR_MAX) can be used to determine an implementation-defined behavior (whether the type **char** uses the same representation as **signed char** or **unsigned char**).

Example

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
if (CHAR_MIN == SCHAR_MIN)
   {
   printf("char appears to be signed\n");
   if (CHAR_MAX != SCHAR_MAX)
      printf("nonconforming implementation\n");
   }
if (CHAR_MIN != SCHAR_MIN)
   printf("char does not appear to be signed\n");
}
```

Otherwise, the value of **CHAR_MIN** shall be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.[15)] 327

Commentary

*_MIN 304 same type as

When the type **char** supports the same range of values as the type **unsigned int**, the type of CHAR_MIN must also be **unsigned int** (i.e., the body of its definition is likely to be 0u).

Coding Guidelines

The possibility that the expression (CHAR_MIN == 0) && (CHAR_MIN <= -1) can be true is likely to be surprising to developers. However, implementations where this macro has type **unsigned int** are not sufficiently common to warrant a guideline recommendation (e.g., perhaps suggesting that CHAR_MIN always be cast to type **int**, or that it not appear as the operand of a relational operator).

Example

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
```

```
6   if (CHAR_MIN == 0)
7      {
8      printf("char appears to be unsigned\n");
9      if ((CHAR_MAX - UCHAR_MAX) != 0)
10        printf("nonconforming implementation\n");
11     }
12
13  if (((int)CHAR_MIN) <= -1) /* Handle the case #define CHAR_MIN 0U */
14     printf("char does not appear to be unsigned\n");
15  }
```

328 The value **UCHAR_MAX** shall equal $2^{\mathbf{CHAR\_BIT}} - 1$. UCHAR_MAX value

Commentary

Any unsigned character types are not allowed to have any hidden bits that are not used in the representation. Therefore UCHAR_MAX must equal $2^{\texttt{sizeof(char)} \times CHAR\_BIT} - 1$. Since `sizeof(char) == 1` by definition, the above C specification must hold.

571 unsigned char pure binary

1124 sizeof char defined to be 1

C90

This requirement was not explicitly specified in the C90 Standard.

C++

Like C90, this requirement is not explicitly specified in the C++ Standard.

Other Languages

Most languages do not get involved in specifying this level of detail.

329 **Forward references:** representations of types (6.2.6), conditional inclusion (6.10.1).

5.2.4.2.2 Characteristics of floating types <float.h>

330 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[16)] floating types characteristics

Commentary

Floating-point types are an alternative to integer representation of numeric values. Some of the floating types occupy the same amount of storage as the larger integer types. Floating-point types trade-off some accuracy representation bits to hold an exponent value. This exponent is used to increase the range of values that can be represented at the cost of a reduced number of significant digits.

The C90 Standard made no mention of ISO/IEC 60599 floating-point arithmetic (although IEEE-754 is mentioned in an example). At that time this standard was only just emerging as the computer industry-wide model of choice. There were still other formats in widespread use. The usage of these formats continues to decline and is now limited to a small number of markets.

In at least one case, it is possible for a program to check which floating-point representation is being used. The __STDC_IEC_559__ macro has the value 1 if the implementation conforms to annex F (IEC 60559 floating-point arithmetic).

2015 __STDC_IEC_ macro

This model assumes its components have a fixed width during program execution. Models based on a variable number of bits in the exponent and significand (adjusted as the value represented gets larger or smaller) have been proposed.[1494] Such variable-width models have yet to be used by commercial processors.

Rationale

The characterization of floating point follows, with minor changes, that of the Fortran standardization Committee. The C89 Committee chose to follow the Fortran model in some part out of a concern for Fortran-to-C translation, and in large part out of deference to the Fortran Committee's greater experience with fine points of floating

point usage. Note that the floating point model adopted permits all common representations, including sign-magnitude and two's-complement, but precludes a logarithmic implementation.

The C89 Committee also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which were contrary to that standard. IEEE 754 is now an international standard, IEC 60559; and that is how it is referred to in C99.

C++

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**

Other Languages

Many languages contain floating-point types. Ada gets involved in very low-level details of the representation model. Fortran (90) contains inquiry functions that return values representing properties of the real types used by an implementation. Lisp defines a bignum package, allowing calculations to precisions greater than that supported by most C implementations **long double**. However, the performance is not as great. One of the design aims of the Java language was to ensure that the behavior of a program did not vary across implementations. While the Java language does specify support for a floating-point standard:

The Java types **float** *and* **double** *are IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point values, respectively.*

it is necessary to remember that this standard permits some degree of implementation leeway. The Java designers attempt to exactly specify how an implementation should follow the floating-point standard (in order to prevent differences in generated results) has met with some criticism.[701]

Common Implementations

Most implementations use the floating-point format supported by the processor on which the program is executed.

Writing a C translator does not require the use of floating-point types. A translator can output floating-point literals in some canonical representation, which is converted to the host representation on program startup (before control is transferred to `main`). However, many translators do make use of knowledge of the representation used at execution time and output, to object files, floating-point literals in this format.

The Unisys e-@ction Application Development Solutions (formerly known as the Universal Compiling System [UCS])[1391] (see Table 330.1) has a word size that is not an integer multiple of the types defined by IEC 60559.

Table 330.1: Range of representable floating-point values for the Unisys e-@ction Application Development Solutions Compiling System.

Type	Bits	Decimal Range
float	36	1.4693680E-39 . . . 1.7014118E+38
double	72	2.7813423E-309 . . . 8.9884657E+307
long double	72	2.7813423E-309 . . . 8.9884657E+307

Some implementations emulate the larger representations by joining together several smaller representations. For instance, the Apple numerics implementation on the PowerPC uses two doubles, each using 64 bits, to represent a **long double**, occupying 128 bits.[52] The PowerPC processor does not support operations on 128-bit floating-point types; they are implemented with some software support. The exponents are adjusted

so that the *least significant* component has an exponent that is at least 54 (representing a value of 2 to 54) less than the *most significant* component. This arrangement increases the precision of the significand (unless it is near the smallest normal value) but does not change the range of possible exponents. The original IBM 390 floating-point format[1207] (base 16) also used two doubles to represent a **long double**. In some cases the sequence of value bits, in the object representation, may be disjoint. For instance, the Motorola 68000 only uses 80 bits of the possible 96 bits in its double extended object representation.[968]

For those applications requiring even greater precision, use of four doubles (providing 212 bits of significand) has been proposed.[572]

Coding Guidelines

Goldberg[500] is frequently recommended reading for developers working with floating-point types. The following provides a rough-and-ready introduction to error analysis.

Assume alg represents the value returned by the C algorithm for the mathematically exact function f, the error is (for simplicity assume a single argument x):

$$error = alg(x) - f(x) \tag{330.1}$$

If alg is a good approximation to f, another way to look at this is to say the result returned by alg at x corresponds to the exact value at the point $x + e$, where e is a very small displacement from x:

$$alg(x) \approx f(x + e) \tag{330.2}$$

Taking the first two terms from a Taylor series expansion about x (assuming f does not contain any discontinuities):

$$f(x + e) \approx f(x) + e * f'(x) \tag{330.3}$$

Combining the preceding three equations we get:

$$\begin{aligned} error &= alg(x) - f(x) && (330.4)\\ &\approx f(x + e) - f(x) && (330.5)\\ &\approx f(x) + e * f'(x) - f(x) && (330.6)\\ &\approx e * f'(x) && (330.7) \end{aligned}$$

showing that the error is proportional to the derivative of the function. To within the approximations used, the algorithm used is not a factor in the error analysis; the mathematical function used to solve the application-domain problem is the important factor.

The formula given in mathematical textbooks usually assume infinite precision and are not well-behaved when less precision is used.[702] A well-known example is computing the area of a triangle using Heron's formula:

$$Area = sqrt(s(s - x)(s - y)(s - z)) \tag{330.8}$$

where: x, y, z are the length of the sides, and $s = (x + y + z)/2$.

For narrow, pointed, triangles this formula can give incorrect results, even when every floating-point operation is correctly rounded. Table 330.2 gives an example where the values are rounded to five significant digits. The final result can either be 0.0 or 1.5813 (the correct area is 1.000025).

Table 330.2: Area of triangle, using Heron's formula, calculated using different rounding directions.

	Correct	Rounding Down	Rounding Up
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x+(y+z))/2$	100.015	100.01	100.02
Area	1.000025	0.0000	1.5813

By recognizing that intermediate results have a finite accuracy, the preceding formula can be reorganized. First the relative values of the length of the sides is important. They need to be sorted so that $x \le y \le z$. The formula:

$$A = sqrt(\frac{(x+(y+z))(z-(x-y))(x+(y-z))}{4}) \tag{330.9}$$

then gives a good, numerically stable approximation to the area. Note that it is only guaranteed to work correctly if the parentheses are not removed (they tell the translator that operands cannot be reordered). On processors that do not support denormal numbers, flushing very small values to zero, this formula fails because a subtraction, $(p-q)$, can underflow. For a more detailed analysis of this problem, see Kahan.[703]

Rev 330.1

Developers who are not familiar with floating-point error analysis shall not be involved in designing or implementing algorithms that use floating-point types.

Usage

Many of the following identifiers were referenced from one program, `enquire.c`, whose job was to deduce the characteristics of a host's floating-point support.

Table 330.3: Number of identifiers defined as macros in **<float.h>** (see Table 770.3 for information on the number of identifiers appearing in the source) appearing in the visible form of the `.c` and `.h` files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
DBL_MIN	9	21	FLT_MAX	5	15	FLT_ROUNDS	18	14
DBL_MAX	20	19	FLT_DIG	5	15	FLT_RADIX	20	14
DBL_DIG	41	17	LDBL_MIN_EXP	4	14	FLT_MIN_EXP	4	14
FLT_EPSILON	4	16	LDBL_MIN	4	14	FLT_MIN_10_EXP	4	14
DBL_MIN_EXP	4	16	LDBL_MIN_10_EXP	4	14	FLT_MAX_EXP	4	14
DBL_MIN_10_EXP	4	16	LDBL_MAX_EXP	4	14	FLT_MAX_10_EXP	4	14
DBL_MAX_EXP	27	16	LDBL_MAX	4	14	FLT_MANT_DIG	8	14
DBL_MAX_10_EXP	14	16	LDBL_MAX_10_EXP	4	14	FLT_EVAL_METHOD	0	0
DBL_MANT_DIG	14	16	LDBL_MANT_DIG	4	14	DECIMAL_DIG	0	0
DBL_EPSILON	4	16	LDBL_EPSILON	4	14			
FLT_MIN	5	15	LDBL_DIG	4	14			

The following parameters are used to define the model for each floating-point type: 331

Commentary

These parameters are not C-language specific. They are the values needed to fully define, mathematically, a floating-point representation. The values in this model are used, by developers, to:

- ensure that implementations support the range of values likely to be needed by an application,
- perform error analysis, and
- calculate the bit pattern representation of hexadecimal floating constants.

floating constant syntax 842

Common Implementations

Most implementations divide the representation of a floating-point type into groups of contiguous sequences of bits, each representing a different component of the model presented here.

---

332 s sign (± 1)

Commentary

One of the few operations that can be performed on a floating-point value, without fear of loss of accuracy, is to change its sign. Information on the sign of the value is held separately from the significand, unlike two's complement notation where it changes the encoding of the value. A consequence of this representation is that it is possible to represent both +0.0 and -0.0 (this possibility also occurs for integers in sign and magnitude representation; and one's complement, but for a different reason). The copysign library function can be used to directly access the sign of a floating-point number.

Common Implementations

Some floating-point formats, e.g., VAX and the HP–was DEC– Alpha in VAX mode, do not support -0.0 (it is considered to be a NaN). While IBM S/360 supports -0.0, any use of it is treated as +0.0. Also -0.0 cannot be generated by normal floating-point operations. The representation of NaNs have a sign bit, but it is not always possible to change this sign bit.

Processors using two's complement floating-point formats have been built in the past. In this representation –MAX is not representable.

---

333 b base or radix of exponent representation (an integer > 1)

Commentary

The FLT_RADIX macro specifies the radix used by an implementation. 366 FLT_RADIX

---

334 e exponent (an integer between a minimum e_{min} and a maximum e_{max}) exponent

Commentary

The range of possible exponent values is governed by the radix and the number of bits used in its representation. Discussions on how many bits of accuracy to trade-off against exponent range have now resolved themselves, or at least the Standard's Committee has published generally agreed-on numbers. The *_MIN_EXP and *_MAX_EXP macros, along with FLT_RADIX define the possible range of values.

Common Implementations

Most implementations use a biased exponent representation (the TMS320C3x[1341] is one of the few that uses two's complement). Here a fixed constant is added to the exponent as it appears in its human-readable form, so the value actually held in the bit pattern representation is always positive (hence the term *biased exponent*; this notation is also known as *excess-N*). By using such a notation for the exponent, it is possible to perform relational comparisons on floating-point values by treating their bit pattern as an *integer* type. This considerably simplifies the hardware implementation of floating-point comparison operations.

Some of the bit patterns representing possible exponent values are also given special meaning by IEC 60559 (e.g., all bits 1 is used to indicate one of several possible states). The Unisys A Series[1389] represents the value of the exponent using 6 bits to represent the magnitude and 1 bit to represent the sign. The CDC 6600[211] and 7600 used an 11-bit one's complement representation for the exponent.

Usage

The range of exponent values that can occur within programs may depend on the application domain. For instance, astronomy programs may contain ranges of very large values and subatomic particle programs contain ranges of very small values. A study of software for automotive control systems[267] showed (see Table 334.1) a relatively small range of exponents, close to zero.

Table 334.1: Dynamic distribution of decimal exponents, as a percentage, for operands of various floating point operations. Adapted from Connors, Yamada, and Hwu[267] (thanks to Connors for supplying the raw data).

Exponent	Compare	Add	Multiply	Divide	Exponent	Compare	Add	Multiply	Divide
0	15.60	11.4	6.7	3.0					
-1	2.5	2.5	1.9	0.0	1	10.80	9.3	1.6	1.0
-2	0.7	1.2	0.6	1.0	2	5.20	2.6	1.3	3.0
-3	0.1	0.0	0.7	0.0	3	8.50	4.3	0.7	0.0
-4	0.0	0.1	0.2	1.0	4	0.50	0.0	0.5	0.0
-5	0.0	0.0	0.5	0.0					
-6	0.0	0.6	1.4	0.0					

precision
floating-point

p precision (the number of base-b digits in the significand) 335

Commentary

The larger the value of p, the greater the precision that can be represented. This value is ultimately responsible for the values of the `*_DIG macros`.

*_DIG 369
macros

The points in the real continuum that can be represented by a floating-point value are not as self-evident as they are for the integer values. Because of the presence of an exponent and a normalized significand, the

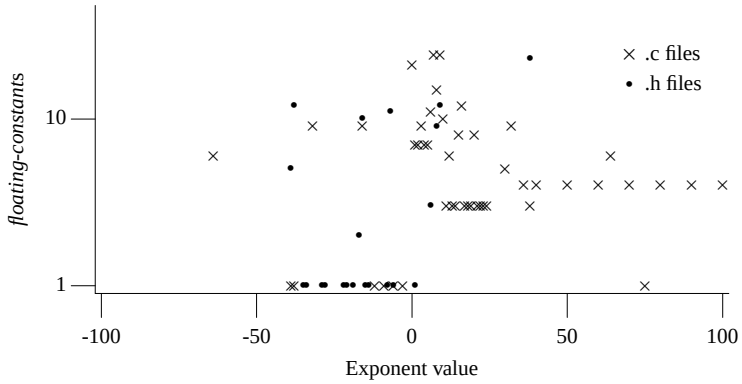


Figure 334.1: Number of *floating-constant*s (that included an *exponent-part*) having a given exponent value. Based on the visible form of the `.c` and `.h` files.

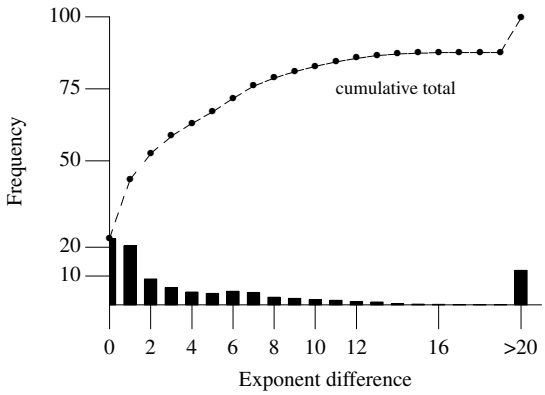


Figure 334.2: Difference in the value of the exponents (in powers of 2) of the two operands of floating-point addition and subtraction operations, obtained by executing the SPECfp92 benchmarks. Adapted from Oberman.[1024]

distance between representable values increases as the numbers get bigger. For a radix of 2, for instance, the number of representable values between each power of two is the same. This means there are as many representable floating-point values between 2 and 4 as there are between 1,048,576 and 2,097,152.

C++

The term *significand* is not used in the C++ Standard.

Common Implementations

The fact that there is a sign bit gives away the representation of the significand in IEC 60559. It is held in sign and magnitude format. This representation was chosen primarily on the grounds of ease of implementation in hardware.

The IEC 60559 representation also uses an additional hidden bit. The bits of the significand are shifted such that the most significant bit that is set, always occurs immediately to the left of the bits actually stored (the value of the exponent is updated to ensure that the actual value being represented remains unchanged), a process called *normalization*. Because this bit is always known to be set, there is no need to store it. Thus there is always one more effective bit of accuracy than is stored in the significand. Implementations that do not use a radix of 2 cannot make use of this hidden-bit technique to obtain more accuracy. For instance, with a radix of 16, bits in the significand need to be shifted by 4 for every incremental change in value of the exponent. The IEC 60559 technique of using a hidden bit can incur a significant performance penalty for software emulations of floating-point operations. Using a radix other than 2 reduces the performance overhead associated with normalizing the significand.[267]

On the WE DSP32[64] the least-significant bit of the significand representation can optionally be used as a parity (odd) bit. Floating-point operations treat this parity bit as part of the value representation. This introduces an error or 1 ULP in 50% of cases.

WE DSP32
346 ULP

The Motorola DSP563CCC[967] uses a 24-bit two's complement representation for the significand (no hidden bit). The CDC 6600[211] and 7600 used a 49-bit one's complement representation for the significand. The radix point was at the least significant end.

Research on hardware support for reusing the results of previously executed sequences of instructions usually requires that values input to an instruction sequence match those of previous evaluations, before the previous computed result can be reused. A study by Álvarev, Corbal, Salamí, and Valero,[21] using multimedia applications (e.g., JPEG encoding), investigated what they called *tolerant reuse*. By ignoring some of the least significant bits (up to four) of floating-point values, they were able to significantly increase the number of previously computed results that could be reused (because the input values matched at lower precisions). The number of instructions executed by programs was reduced by between 25% to 40%.

940 value profiling

A study by Tong, Rutenbar, and Nagle[1356] analyzed the performance of four floating-point intensive applications. They found that it was possible to significantly reduce the number of bits in the significand representation (a speech-recognition program to 5 bits; a fingerprint classification program to 11 bits; an image-processing benchmark to 9 bits; a neural network trainer to 5 bits), without unduly affecting final program accuracy. Customizing the design of floating-point units (i.e., the number of bit in the exponent and significand) to match the accuracy/performance/cost requirements of specific applications has also been proposed.[468]

Coding Guidelines

The term commonly used by developers is *mantissa*, not *significand*. There is probably little to be gained by attempting to change developers' common terminology.

---

336 f_k nonnegative integers less than b (the significand digits)

Commentary

If the radix is 2 (as specified by IEC 60559), possible values for this quantity are 0 and 1.

Common Implementations

Most implementations now follow IEC 60559. The IBM 360 originally used a radix of 16; possible values of f range from 0 to 15 for this implementation. Its successor, the IBM 390, also includes support for the IEC 60559 Standard.[1207]

---

floating-point model

A *floating-point number* (*x*) is defined by the following model: 337

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

Commentary

This defines the term *floating-point number*. This model applies to a broad range of floating-point implementations. It differs from the model for signed magnitude integer types in that it contains an exponent part (which enables the decimal point to *float*) and offers support for a radix other than two.

The presence of a decimal point is not unique to floating-point numbers. Both fixed-point and logarithmic representations provide support for a decimal point. In the fixed-point representation the position of the decimal point in the number is fixed (i.e., only two digits after the decimal point are supported). Support for fixed-point types in C is one of the constructs specified in the embedded C TR. In a logarithmic number system a numeric value is represented by its logarithm. This means that operations, such as multiply and divide, are very fast; however, addition and subtraction are much slower. Recent research,[254] using base 2 logarithms, has shown a factor of two speed improvement (over comparable floating-point implementations) when the ratio of floating-point add to multiple operations was 40% to 60%. A processor using this logarithmic number system has also been built.[255]

Embed-ded C TR 18

C90

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{min} \le e \le e_{max}$$

The C90 Standard did not explicitly deal with subnormal or unnormalized floating-point numbers.

C++

The C++ document does not contain any description of a floating-point model. But, Clause 18.2.2 explicitly refers the reader to ISO C90 subclause 5.2.4.2.2

Coding Guidelines

This model specifies the maximum range of values and precision of a floating-point type. These, and other quantities, are exposed to the developer via macros defined in the header **<float.h>**. The relevant guidelines are covered where these identifiers are discussed.

---

floating types can represent

In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ($x \neq 0$, $e = e_{min}$, $f_1 = 0$) and unnormalized floating-point numbers ($x \neq 0$, $e > e_{min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. 338

Commentary

This terminology and choice of values of x, f_1, and e originates with IEC 60559 (or to be exact IEEE-754 and IEEE-854 standards that eventually became the ISO/IEC Standard). The ability to support subnormal floating-point numbers, unnormalized floating-point numbers, and NaNs is not unique to ISO 60599, although they may be referred to by other names. A terminological note: The 854 Standard uses the term *subnormal* to refer to entities that the 754 Standard calls *denormals* (the term *denormal* and *denormalized* is not only seen in older books and papers but is still in use).

In a normalized floating-point number the value is always held in a form such that the significand has no leading zeros (i.e., the first digit is always one). The exponent is adjusted appropriately. The minimum values for floating-point quantities given elsewhere are required to be normalized numbers.

330 floating types characteristics
subnormal numbers

Subnormal numbers provide support for gradual underflow to zero. The difference in value between the smallest representable normalized number closest to zero and zero is much larger than the difference between the last two smallest adjacent representable normalized numbers. Rounding to zero is thus seen as a big jump. Subnormal numbers populate this chasm with values that provide a more gradual transition to zero. Their representation is such that most significant bits of the significand can be zero, hence the name *subnormal*. As the value moves toward zero, more and more of the significant bits of the significand become zero. There is obviously a decrease in precision for subnormal numbers, the exponent already having its minimum value and there being fewer representable bits available in the significand.

For implementations that support subnormals, the test `x != y` implies that `(x-y) != 0` (which need not be true if subnormals are not supported and enables code such as `if (x != y) z=1/(x-y)` to be written).

Floating-point infinities are used to handle overflows in arithmetic operations. The presence of a sign bit means that implementations often include representations for positive and negative infinity. The infinity values (positive and negative) are not a finite floating-point value. An infinity returned as the result of a subexpression does not necessarily percolate through the remainder of the evaluation of an expression. For instance, a nonzero value divided by infinity returns a zero value.

The `fpclassify` macro returns information on the classification of a floating-point value.

C90

The C90 Standard does not mention these kinds of floating-point numbers. However, the execution environments for C90 programs are likely to be the same as C99 in terms of their support for IEC 60559.

C++

The C++ Standard does not go into this level of detail.

Other Languages

The class `java.lang.Float` contains the members:

```
1  public static final float NEGATIVE_INFINITY = -1.0f/0.0f;
2  public static final float POSITIVE_INFINITY = 1.0f/0.0f;
3  public static final float NaN = 0.0f/0.0f;
```

The class `java.lang.Double` contains the same definitions, but using literals having type **`double`**.

Common Implementations

Handling subnormals in hardware would introduce a lot of complexity and because they are expected to occur very infrequently[700] operations on them are often handled in software (which the hardware traps to when such a value is encountered). In applications where subnormals are quiet common the overhead of software implementation can have a huge impact on performance.[815]

The first digit of a normalized floating-point number is always 1. The IEC 60559 Standard makes use of this property by not holding the 1 in the stored value. An IEC 60559 single-precision, 23-bit significand actually represents 24 bits, because there is an implicit leading digit with value 1. How is zero represented? A significand with all bits zero does not represent a value of zero because of the implicit, leading 1. To represent a value of zero, one of the values of the exponent is required (all bits zero in both the significand and the biased exponent is used).

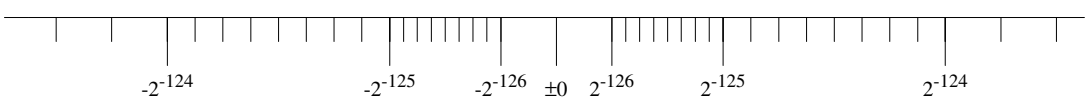


Figure 338.1: Range of normalized numbers about zero, including subnormal numbers.

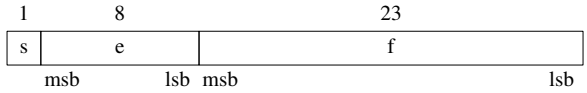


Figure 338.2: Single-precision IEC 60559 format.

guard digit

One of the consequences of having an implicit leading digit is that the significand needs to right-shifted by one digit (and the leading 1 used to fill the vacated bit) before any operations can be performed on it. This shift operation would remove one (binary) digit of accuracy unless the least significant digit was saved. The IEC 60559 Standard specifies that implementations use a so-called *guard digit* to hold this shifted value, preventing accuracy from being lost. Before returning the result of an arithmetic operation, the significand is left-shifted. This guard digit is described here because there are some implementations that do not support it (Cray is well-known for having some processors that don't).

Other information that needs to be maintained by a conforming IEC 60559 implementation, during arithmetic operations on floating-point types, include a *rounding digit* and a *sticky bit*. Carter[206] discusses some of the consequences of implementations that support a subset of this information.

Table 338.1: Format Parameters of IEC 60559 representation. All widths measured in bits. Intel's *extended-precision* format is a conforming IEC 60559 format derived from that standards *extended double-precision* format.

Parameter	Single	Single Extended	Double	Double Extended	Intel x86 Extended
Precision, p, (apparent mantissa width)	24	32	53	64	64
Actual mantissa width	23	31	52	63	64
Mantissa's MS-Bit	hidden bit	unspecified	hidden bit	unspecified	explicit bit
Decimal digits of precision, $p/\log_2(10)$	7.22	9.63	15.95	19.26	19.26
E_{max}	+127	+1023	+1023	+16383	+16383
E_{min}	-126	-1022	-1022	-16382	-16382
Exponent bias	+127	unspecified	+1023	unspecified	+16383
Exponent width	8	11	11	15	15
Sign width	1	1	1	1	1
Format width (9) + (8) + (4)	32	43	64	79	80
Maximum value, $2^{E_{max}+1}$	3.4028E+38	1.7976E+308	1.7976E+308	1.1897E+4932	1.1897E+4932
Minimum value, $2^{E_{min}}$	1.1754E-38	2.2250E-308	2.2250E-308	3.3621E-4932	3.3621E-4932
Denormalized minimum value, $2^{E_{min}-(4)}$	1.4012E-45	1.0361E-317	4.9406E-324	3.6451E-4951	1.8225E-4951

Unnormalized numbers can only occur in the extended formats. No extended formats, known to your author, uses unnormals. While the Intel 80387 (and later) double-extended format allows for unnormals, they are treated as signaling NaNs. The early Intel 80287 (produced before the IEEE-754 standard was finalized) used unnormals. The IBM S/360 supports them, but does not produce them as the result of any normalized floating-point operation. Use of an unnormalized representation can be used to provide an indication of the degree of precision available in a value[62] and reduce the time taken to perform floating-point operations. Unnormalized numbers are usually created by poor quality implementations.

Table 338.2: List of some results of operations on infinities and NaNs. Also see: "Expression transformations" in annex F.8.2 of the C Standard.

$Operation \Longrightarrow Result$	$Operation \Longrightarrow Result$
$x/(+\infty) \Longrightarrow +0$	$x/(+0) \Longrightarrow +\infty$
$x/(-\infty) \Longrightarrow -0$	$x/(-0) \Longrightarrow -\infty$
$(+\infty) + x \Longrightarrow +\infty$	$x + NaN \Longrightarrow NaN$
$(+\infty) \times x \Longrightarrow +\infty$	$\infty \times 0 \Longrightarrow NaN$
$(+\infty)/x \Longrightarrow +\infty$	$0/0 \Longrightarrow NaN$
$(+\infty) - (+\infty) \Longrightarrow NaN$	$NaN - NaN \Longrightarrow NaN$

Some processors do not support subnormal numbers in hardware. A solution sometimes adopted is for an occurrence of such a value to cause a trap to a software routine that handles it. On such processors operations on subnormals can execute significantly more slowly than on normalized values.[316]

- The Cray T90 and IBM S/360 do not handle subnormal numbers in hardware or software. Such a value input to a floating-point functional unit is forced to zero. Underflow results from arithmetic operations are forced to zero. Such an implementation choice is sometimes made to favor performance over accuracy.

- The AMD 3DNow! extensions[350] to the Intel x86 instruction set include support for an IEC 60559 single-precision, floating-point format that does not include NaNs, infinities, or subnormal numbers.

- The Intel SSE extensions[627] to the Intel x86 instruction set include a status bit that, when set, causes subnormal numbers to be treated as zero. Setting this status bit speeds up operations and can be used by those applications where the loss of accuracy is not significant.

- The Motorola DSP563CCC[967] does not support NaNs or infinities. Floating-point arithmetic operations do not overflow to infinity; they saturate at the maximum representable value.

- The HP Precision Architecture[571] supports a quad format (113-bit precision with a 15-bit exponent).

Example

```
1  #include <float.h>
2  #include <stdio.h>
3
4  double x, y;
5
6  int main(void)
7  {
8  double inverse;
9
10 if ((DBL_MIN / 2.0) > 0.0)
11    printf("This implementation supports extended precision or subnormal numbers\n");
12 if ((double)(DBL_MIN / 2.0) > 0.0)
13    printf("This implementation supports subnormal numbers\n");
14
15 if (x != y) /* Only works if subnormals are supported. */
16    inverse = 1.0 / (x - y);
17 }
```

Table 338.3: Example of gradual underflow. * Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Variable or Operation	Value	Biased Exponent	Comment
A0	1.100 1100 1100 1100 1100 1101 × 2^{-125}	2	
A1 = A0 / 2	1.100 1100 1100 1100 1100 1101 × 2^{-126}	1	
A2 = A1 / 2	0.110 0110 0110 0110 0110 0110 × 2^{-126}	0	Inexact*
A3 = A2 / 2	0.011 0011 0011 0011 0011 0011 × 2^{-126}	0	Exact result
A4 = A3 / 2	0.001 1001 1001 1001 1001 1010 × 2^{-126}	0	Inexact*
.			
.			
.			
A23 = A22 / 2	0.000 0000 0000 0000 0000 0011 × 2^{-126}	0	Exact result
A24 = A23 / 2	0.000 0000 0000 0000 0000 0010 × 2^{-126}	0	Inexact*
A25 = A24 / 2	0.000 0000 0000 0000 0000 0001 × 2^{-126}	0	Exact result
A26 = A25 / 2	0.0	0	Inexact*

NaN

A *NaN* is an encoding signifying Not-a-Number.

Commentary

The encoding for NaNs specified in IEC 60559 requires them to have the maximum exponent value (just like infinity) and a nonzero significand. There is no specification for how different kinds of NaN might be represented. In IEC 60559 the expression `0.0/0.0` returns a NaN value.

A NaN always compares unequal to any other value, even the identical NaN. Such a comparison can also lead to an exception being raised.

Rationale

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEC 60559 floating-point standard recommends that NaNs propagate; but it does not require this and not all implementations do. And the floating-point standard fails to specify the contents of NaNs through format conversion. Making signaling NaNs predictable imposes optimization restrictions that anticipated benefits don't justify. For these reasons this standard does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.

Rationale

IEC 60559 (International version of IEEE-754) requires two kinds of NaNs: Quiet NaNs and Signaling NaNs. Standard C only adopted Quiet NaNs. It did not adopt Signaling NaNs because it was believed that they are of too limited utility for the amount of work required.

C90

This concept was not described in the C90 Standard.

C++

Although this concept was not described in C90, C++ does include the concept of NaN.

18.2.1.2p34 Template class numeric_limits

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

18.2.2.1p37

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."[194)]

Other Languages

As standards for existing languages are revised, they are usually updated to support the operations and properties described in IEC 60559.

Common Implementations

Support for NaNs has been available on a variety of hardware platforms and therefore C90 implementations since the late 1980s. Some implementations encode additional diagnostic information in the don't care bits of a NaN representation (e.g., the operation on the Apple PowerPC which created the NaN provides the offset of instruction that created it).

Coding Guidelines

Additional information encoded in a NaN value by an implementation may be of use in providing diagnostic information. The availability of such information and its encoding is not specified by the standard and such usage is making use of an extension.

Example

```
1  double d_max(double val1, double val2)
2  {
3  /*
4   * IEC 60559 requires an exception to be raised if
5   * either of the following operands is a NaN.
6   */
7  if (val1 < val2) /* Result is 0 (false) if either operand is a NaN. */
8     return val2;
9  else
10    return val1;
11 }
```

---

340 A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception;

NaN raising an exception

Commentary

Once a calculation yields a result of quiet NaN all other arithmetic operations having that value as an operand also return a result of quiet NaN. The advantage of the quiet NaN value is that it can be tested for explicitly at known points in a program chosen by the developer. There disadvantage is that the developer has to insert the checks explicitly. A quiet NaN can still cause an exception to be raised. If it is the operand of a relational or equality operator, IEC 60559 requires that an invalid floating-point exception be raised.

C90

The concept of NaN was not discussed in the C90 Standard.

C++

18.2.1.2p34

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

Common Implementations

Most, if not all, implementations represent quiet NaN by setting the most significant bit of a value's significand.

Coding Guidelines

Checking the result of all floating-point operations against NaN enables problems to be caught very quickly, while a lot of context information is available. However, such pervasive checking could complicate the source significantly, may require context information to be passed back through calling functions, and may impact performance.

It is often more practical to view the implementation of some algorithm, involving floating-point values, as an atomic entity from the point of view of NaN handling. In this scenario the implementation of the algorithm does no checking against NaN. However, it is the responsibility of the user of that function, or inlined source code, to check that the input values are valid and to check that the output values are valid. This approach has the advantage of simplicity and efficiency, but the potential disadvantage of loss of context information on exactly where any NaNs were generated.

Designers of third-party libraries need to consider how information is passed back to the caller. A function may be capable of handling the complete range of possible floating-point values, plus the infinities. But NaN is likely to be an invalid input value. Provided that such an input value does not take part in any comparison operations, it may be possible to allow this value to percolate through to the result. If the input value does appear as the operand of a comparison operator, the possibility of a signal being raised means that either the floating-point comparison macros need to be used or an explicit check for NaN needs to be made.

Example

```
1  #include <math.h>
2
3  enum FLT_CMP {cmp_less, cmp_greater, cmp_equal, cmp_nan};
4
5  enum FLT_CMP cmp_float(float val1, float val2)
6  {
7  if (isnan(val1) || isnan(val2))
8     return cmp_nan;
9
10 if (val1 < val2)
11    return cmp_less;
12 if (val1 > val2)
13    return cmp_greater;
14
15 return cmp_equal;
16 }
```

---

a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.[17)] 341

Commentary

A signaling NaN raises an exception at the point it is created. A signaling NaN cannot be generated from any arithmetic operation. Signaling NaNs can only be generated by extensions outside those specified in C99, and then only by the `scanf` and `strtod` functions. A signaling NaN cannot be generated from an initialization of an object with static storage duration (see annex F.7.5). Assignment of a signaling NaN may, or may not, raise an exception. It depends on how the implementation copies the value and the checks made by the host processor when objects are moved.

There is no requirement that any of the floating-point exceptions cause a signal to be raised. It is permissible for an implementation's behavior, when a signaling NaN is triggered, to crash the program (with

no method being provided to prevent this from occurring). However, if floating-point exceptions are turned into signals, the behavior should be equivalent to `raise(SIGFPE)`.

Signaling NaNs have the advantage of removing the need for the developer to insert explicit checks in the source code. Their disadvantage is that they can be difficult to recover from gracefully. The code that generated the exception may be unknown to the function handling the exception, and returning to continue execution within the previous program flow of control can be almost impossible. The state of the abstract machine is largely undefined after the signal is raised and the standard specifies that the behavior is undefined.

The `fegetexceptflag` function returns information that may enable a program to distinguish between an exception raised by a signaling NaN and an exception raised by other floating-point operations.

C++

```
static const bool has_signaling_NaN;
```

18.2.1.2p37

True if the type has a representation for a signaling "Not a Number."[194]

Common Implementations

Most, if not all, implementations represent signaling NaN by a zero in the most significant bit of a value's significand. Some processors (e.g., the Motorola 68000 family[968]) have configuration flags that control whether the signaling NaNs raise an exception.

Coding Guidelines

If a choice is available, is it better to use quiet NaNs or signaling NaNs? They each have their advantages and disadvantages. The exclusive use of signaling NaNs guarantees that, if one is an operand of an operator that treats it as a floating number (except assignment), a signal will be raised. Using the different kinds of NaNs requires use of different control flow structuring of a program. The type of application and how it handles data will also be important factors in selecting which kind of NaN is best suited. The choice, if one is available, of the type of NaN to use is a design issue that is outside the scope of these coding guidelines. Hauser discusses these issues in some detail.[553]

Example

```
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

static jmp_buf start_again;

void handle_sig_NaN(int sig_info)
{
longjmp(start_again, 2);
}

void calculate_something(void)
{ /* ... */ }

int main(void)
{
if (setjmp(start_again))
   printf("Let's start again\n");
signal(SIGFPE, handle_sig_NaN);

calculate_something();
}
```

signed of non-numeric values

An implementation may give zero and non-numeric values (such as infinities and NaNs) a sign or may leave them unsigned. 342

Commentary

This sentence was added by the response to DR #218, which also added the following wording to the Rationale.

Rationale

The committee has been made aware of at least one implementation (VAX and Alpha in VAX mode) whose floating-point format does not support signed zeros. The hardware representation that one thinks would represent `-0.0` is in fact treated as a non-numeric value similar to a NaN. Therefore, `copysign(+0.0,-1.0)` returns `+0.0`, not the expected `-0.0`, on this implementation. Some places that mention (or might have) signed zero results and the sign might be different than you expect:

The complex functions, in particular with branch cuts;

```
ceil()
conj()
copysign()
fmod()
modf()
fprintf()
fwprintf()
nearbyint()
nextafter()
nexttoward()
remainder()
remquo()
rint()
round()
signbit()
strtod()
trunc()
wcstod()
```

Underflow: In particular: `ldexp()`, `scalbn()`, `scalbln()`.

Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored. 343

Commentary

This sentence was added by the response to DR #218.

footnote 15

15) See 6.2.5. 344

Commentary

types 472

Clause 6.2.5 deals with types.

footnote 16

16) The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical. 345

Commentary

Most translators take what they are given in terms of processor hardware support, as the starting point for implementing floating-point arithmetic. This wording is intended to justify just such a decision.

C++

The C++ Standard does not explicitly describe a floating-point model. However, it does include the template class `numeric_limits`. This provides a mechanism for accessing the values of many, but not all, of the characteristics used by the C model to describe its floating-point model.

Common Implementations

Most modern floating-point hardware is based on the IEEE-754 (now IEC 60559) standard. Some supercomputer designers made the design decision of improved performance over accuracy (which was degraded) for some arithmetic operations. Users of such machines tend to be supported by knowledgeable developers who take care to ensure that the final, applications answers are within acceptable tolerances. Users in a mass market have no such backup service and the IEEE floating-point standards have therefore aimed at providing reliable accuracy.

For economic reasons many DSPs implement fixed-point arithmetic rather than a floating-point. In some cases, because of the application-specific nature of the problems they are used in, the parameters (such as implied decimal point) of the fixed-point model do not need to be flexible; different processors can use different parameters. The C99 Standard does not support fixed-point data types, although they are described in a Technical Report.[657]

One solution to implementing floating-point types on processors that support fixed-point types is to convert the source containing floating-point data operations to make calls to a fixed-point library. A tool that automatically performs such a conversion is described by Kum, Kang, and Sung.[785] It works by first monitoring the range of values taken on by a floating-point object. This information is then used to perform the appropriate scaling of values in the transformed source. An analysis of the numerical errors introduced by such a conversion is given in.[1]

The Motorola 68000 processor[968] supports a packed decimal real format. All digits are represented in base 10 and are held one per byte. The 24-byte types consists of a 3-digit exponent, a 17-digit significand, two don't care bytes, a byte holding the two separate sign bits (one for the exponent, the other for the mantissa), and an extra exponent (for overflows that can occur when converting from the extended-precision real format to the packed decimal real format). The Motorola 68000 processor extended precision floating-point type contains 16 don't care bits in its object representation.

Coding Guidelines

The availability of IEC 60559 Standard compatible floating-point support is sufficiently widespread that concern for implementations that don't support it is not considered to be sufficient to warrant a guideline recommendation. Of greater concern is the widespread incorrect belief, among developers, that conformance to IEC 60559 guarantees the same results. This issue is discussed elsewhere. 29 IEC 60559

Testing scripts that operate by differencing the output from a program with its expected output, often show differences between processors when floating-point values are compared. The extent to which these differences are significant can only be decided by developers. A percentage change in a value is often a more important consideration that its absolute change. For small values close to zero it might also be necessary to take into account likely error bounds, which can lead to small values exhibiting a large percentage change that is not significant because the absolute difference is still within the bounds of error.

---

346 The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined , as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library routine in `<stdio.h>`, `<stdlib.h>` and `<wchar.h>`. floating-point operations accuracy

Commentary

Accuracy has two edges— poor accuracy and excessive accuracy. The standard permits (but does not require) an implementation to return as little as one digit of accuracy, after the decimal point, for the result of arithmetic operations. In practice poor levels of accuracy are unlikely to be tolerated by developers. From

the practical point of view, it is possible to achieve accuracies of 1 ULP. A more insidious problem, in some cases, can be caused by additional accuracy; for instance, on a host that performs floating-point operations in some extended precision:

```
1  #include <stdio.h>
2
3  extern double a, b;
4
5  void f(void)
6  {
7  double x;
8
9  x = a + b;
10 if (x != a + b)
11    printf("x != a + b\n");
12 }
```

any extended precision bits will be lost in the first calculation of a+b when it is assigned to x. The result of the second calculation of a+b may be held in a working register at the extended precision and potentially contain additional value bits not held in x, the result of the equality test then being false.

Accuracy does not just relate to individual operations. The operator sequence of multiply followed by addition is sufficiently common that some vendors have created a fused multiply/add instruction. Such fused instructions not only execute more quickly than the sum of two instruction timings, but often deliver results containing greater accuracy (than would have been obtained by executing separate instructions performing the same mathematical operations).

How is the accuracy of a floating-point operation measured? There are two definitions in common use—relative error and units in the last place (ULP).

1. Relative error is the difference between the two values divided by the actual value; for instance, if the actual result of a calculation should be 3.14159, but the result obtained was 3.14×10^0, the relative error is $0.00159/3.14159 \Rightarrow 0.0005$.

ULP

2. If the floating-point number z is approximated by $d.d.d\ldots d\times b^e$ (where b is the base and p the number of digits in the significand), the ULP error is $\mid d.d.d\ldots d-(z/b^e)\mid b^{p-1}$. When a floating-point value is in error by n ULP the number of contaminated (possibly incorrect) digits in that value is $\log_b n$. For IEC arithmetic b is 2 and each contaminated digit corresponds to a bit in the representation of the value. See Muller[979] for a comprehensive discussion. The error in ULPs depends on the radix and the precision used in the representation of a floating-point number, but not the exponent. For instance, if the radix is 10 and there are three digits of precision, the difference between 0.314e+1 and 0.31416e+1 is 0.16 ULPs, the same as the difference between 0.314e+10 and 0.31416e+10. When the two numbers being compared span a power of the radix, the two possible error calculations differ by a factor of the radix. For instance, consider the two values 9.99e2 and 1.01e3, with a radix of 10 and three digits of precision. These two values are adjacent to the value 1.00e3, a power of the radix. If 9.99e2 is the correct value and 1.01e3 is the computed value, the error is 11 ULPs. But, if 1.01e3 is the correct value and 0.999e3 is the computed value, then the error is 1.1 ULPs.

A floating-point operation usually delivers more bits of accuracy than are held in the representation. The least significant digit is the result of rounding those other digits. A possible error of 0.5 ULP is inherent in any floating-point operation. If a variety of calculations all have the same relative error, their error expressed in ULP can vary (*wobble* is the proper technical term) by a factor of b. Similarly, if a set of calculations all have the same ULP, their relative error can vary by a factor of b.

Several programs have been written to test the accuracy of a processor's floating-point operations.[1204] Various mathematical identities can be used, making it possible for these programs to be written in a high-level language (many are written in C) and be processor-independent. For a description of the latest and most

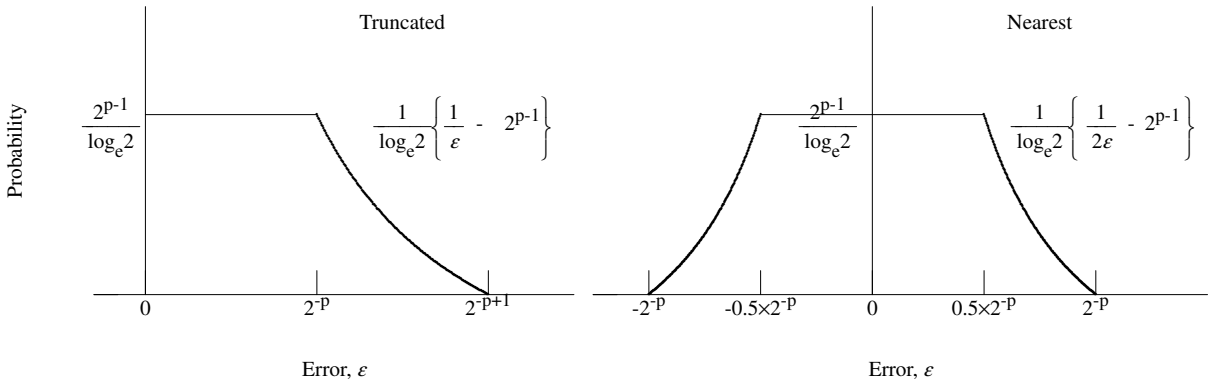


Figure 346.1: Probability of a floating-point operation having a given error (ϵ) for two kinds of rounding modes (truncated and to-nearest); p is the number of digits in the significand. Adapted from Tsao.[778]

thorough testing tool, as of 2001, see Verdonk, Cuyt, and Verschaeren.[1418] For a detailed, mathematical discussion of floating-point accuracy, see Priest.[1123]

The basic formula for error analysis of an operation, assuming $fl(a\ op\ b)$ does not overflow or underflow, is: error analysis

$$fl(a\ op\ b) = (1+\epsilon)\times(a\ op\ b) \tag{346.1}$$

where: $(a\ op\ b)$ is the exact result of the operation, where op is one of $+$, $-$, $*$ and $/$; $fl(a\ op\ b)$ is the floating-point result; $|\ \epsilon\ | \leq macheps$.

To incorporate underflow (but not overflow), we can write:

$$fl(a\ op\ b) = (1+\epsilon) * (a\ op\ b) + \eta \tag{346.2}$$

where:

$$|\ \eta\ | \leq macheps * underflow\ threshold \tag{346.3}$$

On machines that do not implement gradual underflow (including some IEEE machines, which have an option to perform flush-to-zero arithmetic instead), the corresponding error formula is:

$$|\ \eta\ | \leq underflow\ threshold \tag{346.4}$$

(i.e., the error is $1/macheps$ times larger).

Tsao[778] performed an analysis of the distribution of round-off errors, based on Benford's law,[574] to estimate the probability of ϵ having particular values.

Because of the practical difficulty involved in defining a uniform metric that all vendors would be willing to follow (just computing the accuracy reliably could be a significant), and because the importance of floating point accuracy differs greatly among users, the standard allows a great deal of latitude in how an implementation documents the accuracy of the real and complex floating point operations and functions. Rationale

. . .

If an implementation documents worst-case error, there is no requirement that it be the minimum worst-case error. That is, if a vendor believes that the worst-case error for a function is around 5 ULPs, they could document it as 7 ULPs to be safe.

The Committee could not agree on upper limits on accuracy that all conforming implementations must meet, for example, "addition is no worse than 2 ULPs for all implementations.". This is a quality-of-implementation issue.

Implementations that conform to IEC 60559 have one half ULP accuracy in round-to-nearest mode, and one ULP accuracy in the other three rounding modes, for the basic arithmetic operations and square root. For other floating point arithmetics, it is a rare implementation that has worse than one ULP accuracy for the basic arithmetic operations.

. . .

The C99 Committee discussed the idea of allowing the programmer to find out the accuracy of floating point operations and math functions during compilation (say, via macros) or during execution (with a function call), but neither got enough support to warrant the change to the Standard. The use of macros would require over one hundred symbols to name every math function, for example, `ULP_SINF`, `ULP_SIN`, and `ULP_SINL` just for the real-valued sin function. One possible function implementation might be a function that takes the name of the operation or math function as a string, `ulp_err("sin")` for example, that would return a double such as 3.5 to indicate the worst case error, with `-1.0` indicating unknown error. But such a simple scheme would likely be of very limited use given that so many functions have accuracies that differ significantly across their domains. Constrained to worst-case error across the entire domain, most implementations would wind up reporting either unknown error or else a uselessly large error for a very large percentage of functions. This would be useless because most programs that care about accuracy are written in the first place to try to compensate for accuracy problems that typically arise when pushing domain boundaries; and implementing something more useful like the worst case error for a user-specified partition of the domain would be excessively difficult.

DECIMAL_DIG macro

Some of the issues of how representation used can impact accuracy is discussed elsewhere. The issue of the accuracy of decimal string to/from *binary* (non-decimal) floating-point conversions was raised by DR #211. The resolution of this DR resulted in the change of wording highlighted in the preceding C99 sentence.

C90

In response to DR #063 the Committee stated (while the Committee did revisit this issue during the C99 revision of the C Standard, there was no change of requirements):

DR #063

Probably the most useful response would be to amend the C Standard by adding two requirements on implementations:

Require that an implementation document the maximum errors it permits in arithmetic operations and in evaluating math functions. These should be expressed in terms of "units in the least-significant position" (ULP) or "lost bits of precision."

Establish an upper bound for these errors that all implementations must adhere to. The state of the art, as the Committee understands it, is:

correctly rounded results for arithmetic operations (no loss of precision)

1 ULP for functions such as sqrt, sin, and cos (loss of 1 bit of precision)

4–6 ULP (loss of 2–3 bits of precision) for other math functions.

Since not all commercially viable machines and implementations meet these exacting requirements, the C Standard should be somewhat more liberal.

The Committee would, however, suggest a requirement no more liberal than a loss of 3 bits of precision, out of kindness to users. An implementation with worse performance can always conform by providing a more

conservative version of **<float.h>**, *even if that is not a desirable approach in the general case. The Committee should revisit this issue during the revision of the C Standard.*

C++

The C++ Standard says nothing on this issue.

Common Implementations

The accuracy of floating-point operations is usually at the mercy of the host floating-point processor hardware. Many implementations try to achieve an accuracy of 1 ULP in their floating-point operations and **<math.h>** 346 ULP library functions.

Use of the logarithmic number system, to represent real numbers, enables multiplication and division to be performed with no rounding error.[254]

There is an Open Source implementation of the IEEE-754 standard available in software.[552] This provides floating-point operations to a known, high level of accuracy, but somewhat slowly.

In IEC 60559 arithmetic the floating-point result $fl(a\ op\ b)$ of the exact operation $(a\ op\ b)$ is the nearest floating-point number to $(a\ op\ b)$, breaking ties by rounding to the floating-point number whose least significant bit is zero (an *even* number). Taking underflow into account requires an additional term to be added to the right side of the basic equation given earlier. For processors that make use of gradual underflow the value is bounded: $\mid \epsilon \mid \leq macheps \times underflow\_threshold$. On machines that do not implement gradual underflow (including some IEC 60559 base processors, which have an option to perform flush-to-zero arithmetic instead), the corresponding error formula is:

$$\mid \epsilon \mid \leq underflow\ threshold \tag{346.5}$$

Cray had a line of processors (the C90 and Cray 2) on which speed of execution was given priority over accuracy. For this processor the error analysis formula for addition and subtraction was:

$$fl(a \pm b) = ((1 + \epsilon_1) * a) \pm ((1 + \epsilon_2) * b) \tag{346.6}$$

Subtracting two values that are very close to each other can lead to a much larger than normal, in IEC 60559, error. On some Cray processors divide was emulated by multiplying by a reciprocal, leading to $x/x \neq 1.0$ in some cases.

The AMD 3DNow! extensions[350] to the Intel x86 processor include a reciprocal instruction that delivers a result accurate to 14 bits (instead of the normal 23 bits). Developers can choose, with suitable translator support, to accept this level of accuracy or to have two additional instructions generated which return a result more slowly, but is accurate to 1 ULP.

The accuracy of the trigonometric functions $\sin$, $\cos$, and $\tan$ depends on how well their argument is reduced, modulo $\pi/2$, and by the approximations used on the small domain near zero. Some implementations do a very poor job of argument reduction, so values near a multiple of $\pi/2$ often have few, if any, bits correct.[1009]

The Intel x86 processor family contains an instruction for computing the sine (and cosine and tangent) of its operand. This instruction requires the absolute value of its operand, in radians, to be less than 2^{63}. Values outside this range are returned as the result of the instruction and a processor flag, C2, is set. Such a result is a surprise in the sense that `sin` (and sine and tangent) is expected to return a value between -1 and 1.

Another accuracy problem with the Intel x86 instruction for computing the sine (and cosine) is accuracy near π ($\pi/2$ for cosine). When using 64-bit mode the 53-bit significand result may only be accurate in the first 15 bits and the 80-bit mode the 64-bit significand result may only be accurate in the first 5 bits (the

problem is caused by insufficient accuracy, 68 bits, in the approximation of π used; 126 bits are actually needed, and used by AMD in their x86 processors).

Given the difficulty in performing mathematical error analysis on algorithms, let alone coded implementations running on imperfect hardware, one solution is to have the translator generate machine code to estimate round-off errors during program execution. One such translator, Fortran-based, has been developed.[130]

double rounding

Processors that perform extended-based arithmetic operations do not always produce more accurate results. There is a source of error known as *double rounding* that occurs when the rounding mode is round-to-nearest. In the default precision mode, an extended-based system will initially round the result of an arithmetic operation to extended double-precision. If that result needs to be stored in double-precision, it has to be rounded again. The combination of these two rounding operations can yield a value that is different from what would have been obtained by rounding the result of the arithmetic operation directly to double-precision. This can happen when the result as rounded, to extended double-precision, is a *halfway case* (i.e., it lies exactly halfway between two double-precision numbers) so the second rounding is determined by the round-ties-to-even rule. If this second rounding rounds in the same direction as the first, the net rounding error will exceed half a unit in the last place. It can be shown[422] that the sum, difference, product, or quotient of two p-bit numbers, or the square root of a p-bit number, rounded first to q bits and then to p bits gives the same value as rounding the result of the operation just once to p bits provided $q \geq 2p + 2$.

Double-rounding can also prevent execution-time measurement of rounding errors in calculations. The formula:

```
1  t = s + y;
2  e = (s - t) + y;
```

recovers the roundoff error in computing `t`, provided double-rounding does not occur.

Coding Guidelines

There are three sources of inaccuracies in the output from a program that uses floating-point data types. Those caused by the implementation, those intrinsic to the algorithm used, and those caused by how the developer coded the algorithm. For instance, assume that `raise_to_power(valu, power)` is a function that returns `valu` raised to `power` (e.g., `raise_to_power(3.0, 2)` returns 9.0). If we want to raise a value to a negative power, the obvious solution is to use `raise_to_power(1.0/valu, power)`. However, there in a rounding error introduced by the calculation `1.0/valu` that will be compounded each time it is used inside the library function. The expression `1.0/raise_to_power(valu, power)` avoids this compounding of the rounding error; it has a single rounding error introduced by the final divide.

Minimizing the rounding error in floating-point calculations can require some theoretical knowledge. For instance, the simple loop for summing the elements of an array:

```
1  double sum_array(int num_elems, double a[num_elems])
2  {
3  double result = 0.0;
4
5  for (int a_index=0; a_index < num_elems; a_index++)
6     result += a[a_index];
7
8  return result;
9  }
```

can have the rounding error in the final result significantly reduced by replacing the body of the function by the Kahan summation formula:[500]

```
1  double sum_array(int num_elems, double a[num_elems])
2  {
3  double result = a[0],
4         C = 0.0,
```

```
        T, Y;

for (int a_index=1; a_index < num_elems; a_index++)
   {
   Y = a[a_index] - C;
   T = result + Y;
   C = (T - result) - Y;
   result = T;
   }

return result;
}
```

which takes account of the rounding properties of floating-point operations to deliver a final result that contains significantly less rounding error than the simple iterative addition. The correct operation of this summation formula also requires the translator to be aware that algebraic identities that might apply to integer objects, allowing most of the body of the loop to be optimized away, do not apply to floating-point objects.

Usage

In theory it is possible to measure the accuracy required/expected by an application. However, it is not possible to do this automatically — it requires detailed manual analysis. Consequently, there are no usage figures for this sentence (because no such analyses have been carried out by your author for any of the programs in the measurement set).

---

347 The implementation may state that the accuracy is unknown.

Commentary

It is not always possible for an implementation to give a simple formula for the accuracy of its floating-point arithmetic operations, or the accuracy of the **<math.h>** functions. A variety of different implementation techniques may be applied to each floating-point operator[1024, 1035] and to the **<math.h>** functions. The accuracy in individual cases may be known. But a complete specification for all cases may be sufficiently complex that a vendor chooses not to specify any details.

C++

The C++ Standard does not explicitly give this permission.

Other Languages

Most languages don't get involved in this level of detail. However, Ada specifies a sophisticated model of accuracy requirements on floating point operations.

Common Implementations

No vendors known to your author state that the accuracy is unknown.

Coding Guidelines

How such a statement in an implementation's documentation should be addressed is a management issue and is outside the scope of these guidelines.

---

348 All integer values in the **<float.h>** header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives;

float.h suitable for #if

Commentary

This is a requirement on the implementation. It is more restrictive than simply requiring an integer constant expression. The cast and **sizeof** operators cannot appear in **#if** preprocessing directives, although they can appear within a constant expression.

FLT_ROUNDS may be an expression whose value varies at execution time, through use of the `fesetround` library function.

C90

Of the values in the **<float.h>** *header, FLT_RADIX shall be a constant expression suitable for use in* **#if** *preprocessing directives;*

C99 requires a larger number of values to be constant expressions suitable for use in a **#if** preprocessing directive and in static and aggregate initializers.

C++

The requirement in C++ only applies if the header **<cfloat>** is used (17.4.1.2p3). While this requirement does not apply to the contents of the header **<float.h>**, it is very likely that implementations will meet it and no difference is flagged here. The namespace issues associated with using **<cfloat>** do not apply to names defined as macros in C (17.4.1.2p4)

17.4.4.2p2 *All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in* **#if** *preprocessing directives, unless explicitly stated otherwise.*

The C++ wording does not specify the C99 Standard and some implementations may only support the requirements specified in C90.

Common Implementations

This statement was true of some C90 implementations. But, there were many implementations that defined the *_MAX macros in terms of external variables, so the *_MAX macros were not usable in static or aggregate initializers. Exact figures are hard to come by, but perhaps 50% of C90 implementations did not use constant expressions.[1379]

Coding Guidelines

It will take time for implementations to migrate to C99. The extent to which a program relies on a C99 translator being available is outside the scope of these coding guidelines.

Example

```
1  #include <float.h>
2
3  #if FLT_RADIX == 10 /* A suitable constant in C90 and C99. */
4  #endif
5
6  #if FLT_DIG == 5 /* Not required to be conforming in C90. */
7  #error This is not a C99 conforming implementation
8  #endif
```

---

all floating values shall be constant expressions. 349

Commentary

This is a requirement on the implementation. It ensures that floating values can be used as initializers for objects defined with static storage duration.

address constant 1341

C90

all other values need not be constant expressions.

This specification has become more restrictive, from the implementations point of view, in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Common Implementations

This C99 requirement was met by many C90 implementations.

Coding Guidelines

It will take time for implementations to migrate to C99. The extent to which a program relies on a C99 translator being available is outside the scope of these coding guidelines.

---

350 All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types.

Commentary

DECIMAL_DIG applies to " . . . the widest supported floating type . . . ". One of the floating types has to be selected; there cannot be three widest types. It is the type **long double**, even if an implementation supports a floating type with more precision as an extension (the standard requires it to be one of the types it specifies, and is silent on the issue of additional floating-point types provided by the implementation, as an extension).

The fact that the three FLT_* macros have a single value for all three floating-point types implies a requirement that the floating-point evaluation method, radix, and rounding behavior be the same for these floating-point types, which is also required in IEC 60559.

C90

Support for DECIMAL_DIG and FLT_EVAL_METHOD is new in C99. The FLT_EVAL_METHOD macro appears to add functionality that could cause a change of behavior in existing programs. However, in practice it provides access to information on an implementation's behavior that was not previously available at the source code level. Implementations are not likely to change their behavior because of this macro, other than to support it.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Common Implementations

For those processors containing more than one floating-point unit (e.g., Intel Pentium with SSE extensions,[626] AMD Athlon with 3Dnow!,[350] and PowerPC with AltiVec support[971]) implementations may chose to evaluate some expressions in different units. Such implementations may then support more than one name/value for some of these macros; for instance, the evaluation method each representing the characteristics of a different floating-point unit.

1147 multiply always truncate

---

351 The floating-point model representation is provided for all values except **FLT_EVAL_METHOD** and **FLT_ROUNDS**.

Commentary

That is, the characteristics of the model used to define floating types does not include FLT_EVAL_METHOD or FLT_ROUNDS as one of its parameters. The standard lists the possible values for these macros and their meaning separately. The actual values used are implementation-defined. These two macros correspond to two classes of implementation-defined behavior described by the IEC 60559 Standard.

330 floating types characteristics
354 FLT_EVAL_ME
352 FLT_ROUNDS

The IEC 60559 Standard also allows implementations latitude in how some constructs are performed. These macros give software developers explicit control over some aspects of the evaluation of arithmetic operations.

C90

Support for FLT_EVAL_METHOD is new in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Coding Guidelines

Educating developers about the availability of these options and there implications is outside the scope of these coding guidelines. A statement of the form "It is possible to obtain different results, using the same expression, operand values and compiler, between different IEC 60559 implementations or even the same implementation operating in a different mode." might be used as an introductory overview of floating point.

FLT_ROUNDS The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:[18)] 352

-1 indeterminable
0 toward zero
1 to nearest
2 toward positive infinity
3 toward negative infinity

All other values for **FLT_ROUNDS** characterize implementation-defined rounding behavior.

Commentary

One form of rounding that would be classified as indeterminable is known as ROM-rounding.[781] Here a small number of the least significant bits of the value are used as an index into a table (usually held in ROM on the hardware floating-point unit). The output from this table lookup is appended to the other digits to give the rounded result. In the boundary case where there would normally be a ripple add up through to the next significant bit the result is the same as a round toward zero (thus simplifying the hardware and possible improving performance at the cost of an inconsistent rounding operation for a small percentage of values).

Rounding to zero, also known as *chopping*, occurs when floating-point values are converted to integers. Of the methods listed, it generates the largest rounding error in a series of calculations.

Rounding to nearest has minimum rounding error, of the known methods, in a series of calculations. The C Standard does not specify the result in the case where there are two nearest representable values. The IEC 60559 behavior is to round to the even value (i.e., the value with the least significant bit set to zero).

Rounding to positive and negative infinity is used to implement interval arithmetic. In this arithmetic, values are not discrete points; they are represented as a range, with a minimum and maximum limit. Adding two such intervals, for instance, generates another interval whose lower and upper bounds delimit the possible range of values represented by the addition. Calculating the lower bound requires rounding to negative infinity, the upper bound requires rounding to positive infinity.

FLT_ROUNDS is not required to be an lvalue. It could be an internal function that is called. It is generally assumed, but not required, that the rounding mode used for other floating-point operators is the same as that for addition.

The FENV_ACCESS pragma may need to be used in source that uses the FLT_ROUNDS macro. The value of a particular occurrence of the FLT_ROUNDS macro is only valid at the point at which it occurs. The value of this macro could be different on the next statement (or even within the same expression if it contains multiple calls to the `fesetround` library function).

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

The C++ header **<limits>** also contains the enumerated type:

18.2.1.3

```
namespace std {
  enum float_round_style {
    round_indeterminable       = -1,
    round_toward_zero          = 0,
    round_to_nearest           = 1,
```

```
        round_toward_infinity      = 2,
        round_toward_neg_infinity  = 3
      };
    }
```

which is referenced by the following member, which exists for every specialization of an arithmetic type (in theory this allows every floating-point type to support a different rounding mode):

18.2.1.2p62

Meaningful for all floating point types.

```
static const float_round_style round_style;
```

The rounding style for the type.[206)]

Other Languages

Most other language specifications say nothing on this subject. Java specifies round-to-nearest for floating-point arithmetic.

Common Implementations

On most implementations the rounding mode does not just affect addition. It is applied equally to all operations. It can also control how overflow is handled. Some applications may choose a round-to-zero rounding behavior because overflow then returns a saturated result, not infinity. Most implementations follow the IEC 60559 recommendation and use *to nearest* as the default rounding mode.

698 floating value converted not representable

Almost all processors (the INMOS T800[622] does not) allow the rounding mode of floating-point operations to be changed during program execution (often using two bits in one of the processor control registers). The HP–was DEC– Alpha processor[1247] includes both instructions that take the rounding mode from a setable control register and instructions that encode the floating-point rounding mode within their bit-pattern (a translator can generate either form of instruction). The translator for the Motorola DSP563CCC[967] only supports *round-to-nearest*.

Some of the older Cray processors make use of implementation-defined rounding modes for some operators, resulting in greater than 1 ULP errors.

The HP–was DEC– VAX processor[284] has two rounding modes. Round-to-nearest, with the tie break being to round to the larger of the absolute values (i.e., away from zero, as does the HP 2000 series). The other is called *chopped* (i.e., round toward zero). The HP–was DEC– Alpha processor[1247] contains instructions that emulate this behavior for compatibility with existing code.

Some processors have two sets of floating-point instructions. For instance AMD Athlon processors containing the 3DNow! extensions[350] have instructions that perform operations on IEC 60559 compatible single and double floating-point values. The extensions operate on a single-precision type only and do not support full IEC 60559 functionality (only one rounding mode— underflow saturates to the minimum normalized value or negative infinity; overflow saturates to the maximum normalized value or positive infinity; no support for infinities or NaN as operands). The advantage of using these floating-point instructions is that they operate on two sets of operands at the same time (each operand is two 32-bit floating-point values packed into 64 bits), offering twice the number-crunching rate for the right kind of algorithms.

1147 multiply always truncate

Coding Guidelines

Applications that need to change the rounding mode are rare. For instance, a set of library functions implementing interval arithmetic[229,716] needs to switch between rounding to positive and negative infinity. The C Standard lists the programming conventions that its model of the floating-point environment is based on (in the library section). These conventions can be used by implementation's to limit the number of possible different permutations of floating-point status they need to correctly handle.

Example

Table 352.1: Effect of rounding mode (FLT_ROUNDS taking on values 0, 1, 2, or 3) on the result of a single precision value (given in the left column).

	0	1	2	3
1.00000007	1.0	1.00000012	1.00000012	1.0
1.00000003	1.0	1.0	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012

floating operands evaluation format

~~The~~ Except for assignment and cast (which remove all extra range and precision), the values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. 353

Commentary

Operands of both the arithmetic, comparison, and relational operators are subject to the usual arithmetic conversions. Operations that do not cause their operands to be the subject of the usually arithmetic conversions are assignment (both through the assignment operator and the passing of a value as an argument) and casts.

A floating-point expression has both a semantic type (as defined by the usual arithmetic conversion rules) and an evaluation format (as defined by the FLT_EVAL_METHOD). The need for an evaluation format arises because of historical practice and because of how floating-point arithmetic is implemented on some processors. Including floating constants in this list means that, for instance, `0.1f` will be represented using **float** only if FLT_EVAL_METHOD is 0; one guaranteed way of obtaining a **float** representation of this value is:

FLT_EVAL_METHOD [354]

```
1   static const float tenth_f = 0.1f;
```

However, this is not a floating constant (in C, it is in C++).

The wording was changed by the response to DR #290.

C90

6.2.1.5 *The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type;*

This wording allows wider representations to be used for floating-point operands and expressions. It could also be interpreted (somewhat liberally) to support the idea that C90 permitted floating constants to be represented in wider formats where the usual arithmetic conversions applied.

Having the representation of floating constants change depending on how an implementation chooses to specify FLT_EVAL_METHOD is new in C99.

C++

Like C90, the FLT_EVAL_METHOD macro is not available in C++.

Other Languages

Java requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision.

Most other language specifications do not get involved in this level of detail.

Common Implementations

Early implementations of C evaluated all floating-point expressions in double-precision. This practice has continued in some implementations because developers have become used to the extra precision in calculations (also some existing code relies on it).

floating-point architectures

Processor designers have taken a number of different approaches to the architecture of hardware floating-point units.

- *Extended based.* Here all floating-point operands are evaluated in an extended format. For instance, the Intel x86 operates on an 80-bit extended floating-point format. Other processors whose arithmetic engine operates on an extended format include the Motorola 6888x and Intel 960.
- *Double based.* Here floating-point operands are evaluated in the double format. Processors whose arithmetic engine operates on double format includes IBM RS/6000 and Cray (which has no IEC 60559 single format at all).
- *Single/double based.* Here the processor instructions operate on floating-point values according to their type. Operations on the single format are often faster than those on double. Processors whose arithmetic engine operates like this include MIPS, SUN SPARC, and HP PA-RISC.
- *Single/double/extended based.* Here the processor provides instructions that can operate on three different floating-point types. Processors whose arithmetic engine operates like this include the IBM/370.

Coding Guidelines

The existence of evaluation formats are a cause for concern (i.e., they are a potential cost). Porting a program, which uses floating-point types, to an implementation that is identical except for a difference in `FLT_EVAL_METHOD` can result in considerably different output.

64 correctly rounded result

Many floating-point expressions are exact, or correctly rounded, when evaluated using twice the number of digits of precision as the data. For instance:

```
1  float cross_product(float w, float x, float y, float z)
2  {
3  return w * x - y * z;
4  }
```

yields a correctly rounded result if the expression is evaluated to the range and precision of type **`double`** (and the type **`double`** has at least twice the precision of the type **`float`**). By a happy coincidence, the developer may have obtained the most accurate result. Writing code that has less dependence of happy coincidences requires more effort:

```
1  float cross_product(float w, float x, float y, float z)
2  {
3  return ((double)w) * x - ((double)y) * z;
4  }
```

These coding guidelines are not intended as an introductory course on the theory and practicalities of coding floating-point expressions. But, it is hoped that readers will be sufficiently worried by the issues pointed out to either not use floating-point types in their programs or to become properly trained in the subject.

Example

```
#include <tgmath.h>

static float glob;

float f(void)
{
/*
 * If FLT_EVAL_METHOD is zero, we want to invoke the float version of
 * the following function to avoid the double rounding that would occur
 * if the double sqrt function were invoked.
 */
return hypot(+glob, 0.0);
}
```

FLT_EVAL_METHOD

The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:[19)] 354

Commentary

This specification is based on existing practice and provides a mechanism to make evaluation format information available to developers (many C90 implementations were already using the concept of an evaluation format, even though that standard did not explicitly mention it)— existing practice is affected by processor characteristics. Some processors do not provide hardware support to perform some floating-point operations in specific formats. This means that either an available format is used, or additional instruction is executed to convert the result value (slowing the performance of the application).

Demmel and Hida 1163

The actual types denoted by the typedef names `float_t` and `double_t` are dependent on the value of the `FLT_EVAL_METHOD` macro. Demmel and Hida give an error analysis for summing a series when the intermediate results are held to greater precision than the values in the series.

C90

Support for the `FLT_EVAL_METHOD` macro is new in C99. Its significant attendant baggage was also present in C90 implementations, but was explicitly not highlighted in that standard.

C++

Support for the `FLT_EVAL_METHOD` macro is new in C99 and it is not available in C++. However, it is likely that the implementation of floating point in C++ will be the same as in C.

Other Languages

Most language implementations use the support provided by their hosts' floating-point unit, if available. They may also be influenced by the representation of floating types used by other language implementations (e.g., C). Even those languages that only contain a single floating-point type are not immune to evaluation format issues, they simply have fewer permutations to consider. Fortran has quite a complex floating-point expression evaluation specification. This also includes a concept called *widest-need*.

Common Implementations

Implementations often follow the default behavior of the hardware floating point unit, if one is available. In some cases the time taken to perform a floating-point operation does not depend on the evaluation format.[627,968] In these cases the widest format is often the default. When floating-point operations have to be emulated in software, performance does depend on evaluation format and implementations often choose the narrowest format.

The Intel IA64[631] provides a mechanism for software emulation, as does the HP–was DEC– Alpha.[1247]

Coding Guidelines

Calculations involving floating-point values often involve large arrays of these values. Before the late 1990s, the availability of storage to hold these large arrays was often an important issue. Many applications used arrays of floats, rather than arrays of doubles, to save storage space. These days the availability of storage is rarely an important issue in selecting the base type of an array.

The choice of floating-point type controls the precision to which results are rounded when they are assigned to an object. Depending on the value of FLT_EVAL_METHOD, this choice may also have some influence on the precision to which operations, on floating-point operands, are performed.

Example

```
#include <float.h>

#if FLT_EVAL_METHOD < 1
#error Program won't give the correct answers using this implementation
#endif
```

355-1 indeterminable;

Commentary

Some processors support more than one evaluation format and provide a mechanism for encoding the choice within the instruction encoding (rather than in a separate configuration register), and some support both mechanisms.[1247] A program executing on such a processor, may have been built from two translation units—one that was translated using one evaluation format and one using a different evaluation format.

Common Implementations

An example of a processor where a translator might want to support a FLT_EVAL_METHOD macro value of less than zero is the Acorn ARM processor. The ARM machine code generated for the following expression statement:

```
float a;
double b, c;
a = b + c;
```

might be (if the FLT_EVAL_METHOD macro had a value of 0 or 1):

```
adfd    f0, f0, f1   # f0 = f0 + f1, rounded to double
mvfs    f0, f0       # f0 = f0, rounded to single
```

however, the following would be a more efficient sequence:

```
adfs    f0, f0, f1   # f0 = f0 + f1, rounded to single
```

but the addition operation is performed to less precision than is available in the operands. An implementation can only generate this instruction sequence if the FLT_EVAL_METHOD macro has a value less than zero.

Coding Guidelines

At first sight, having to use an implementation that specified a value of -1 for FLT_EVAL_METHOD would appear to be very bad news. It means that developers cannot depend on any particular evaluation format being used consistently when a program is translated. In this case, the responsible developer is forced to ensure that explicit casts are used before and after every arithmetic operation. Casting an operand before an operation guarantees a minimum level of precision. Casting the result of an operation ensures that any extra

precision is not passed on to the next evaluation. The end result is a program that gives much more consistent results when ported.

The extent to which an indeterminable value will affect the irresponsible developer cannot be estimated (i.e., will the differences in behavior be significant; will an indeterminate value for FLT_EVAL_METHOD have made any difference compared to some other value?) For source written by that developer, perhaps not. But the developer may have copied some code from an expert who assumed a particular evaluation method. There is little these coding guidelines can do to address the issue of the irresponsible developer. Given that implementations using an indeterminate value appears to be rare a guideline recommendations is not considered worthwhile.

---

0 evaluate all operations and constants just to the range and precision of the type; 356

Commentary

usual arithmetic conversions 706

In many contexts, operands will have been subject to the usual arithmetic conversions, so objects having type **float** will have been converted to type **double**. The difference between this and an evaluation format value of 1 is that constants may be represented to greater precision. In:

```
1  float f1, f2;
2  if (f1 < f2)
```

as-if rule 122

an implementation may choose to perform the relational test without performing the usual arithmetic conversions by invoking the as-if rule. No recourse to the value of FLT_EVAL_METHOD is needed. But in:

```
1  float f1;
2  if (f1 < 0.1f)
```

the value of FLT_EVAL_METHOD needs to be taken into account because of a possible impact on the representation used for `0.1f`.

Other Languages

This might be thought to be the default specification for many languages that do not aim to get as close to the hardware as C does. However, few languages define the behavior of operations on floating-point values with the precision needed to make this the only implementation option. Performance of the translated program is an issue for all languages and many follow, like C, the default behavior of the underlying hardware (even Fortran).

Common Implementations

This form of evaluation is most often used by implementations that perform floating-point operations in software; it being generally felt that the execution-time penalty of using greater precision is not compensated for by improvements in the accuracy of results. (If a developer wants the accuracy of **double**, let them insert explicit casts).

This evaluation format is not always used on hosts whose floating-point hardware performs operations in **float** and **double** types. The desire for compatibility with behavior seen on other processors may have a higher priority. Some translators provide an option to select possible evaluation modes.

Coding Guidelines

This evaluation format is probably what the naive developer thinks occurs in all implementations. Developers who have been using C for many years may be expecting the behavior specified by an evaluation format value of 1. As such, this evaluation format contains the fewest surprises for a developer expecting the generated machine code to mimic the type of operations in the source code. An implementation is doing exactly what it is asked to do.

---

footnote 17

17) IEC 60559:1989 specifies quiet and signaling NaNs. 357

Commentary

The IEEE 754R committee is tentatively planning to remove the requirement of support for signaling NaNs from the floating-point standard and to not recommend such support. The collected experience of the 754R committee is that use of signaling NaNs has been vanishingly small and in most of those cases alternate methods would suffice. The minimal utility does not justify the considerable cost required of system and tool implementors to support signaling NaNs in a coherent manner nor the cost to users in dealing with overburdened tools and overly complicated specification.

Liaison report to WG14, Sep 2002

---

358 For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

Commentary

The standard does not require that such encoding be provided, only that, if there are encodings with similar behavior, the terms quiet NaN and signaling NaN can be applied to them.

C90

The concept of NaN is new, in terms of being explicitly discussed, in C99.

C++

```
static const bool has_quiet_NaN;
```

18.2.1.2p34

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

Meaningful for all floating point types.

Shall be true for all specializations in which `is_iec559 != false`.

```
static const bool has_signaling_NaN;
```

18.2.1.2p37

True if the type has a representation for a signaling "Not a Number."194)

Meaningful for all floating point types.

Shall be true for all specializations in which is_iec559 != false.

```
static const bool is_iec559;
```

18.2.1.2p52

True if and only if the type adheres to IEC 559 standard.[201)]

The C++ Standard requires NaNs to be supported if IEC 60559 is supported, but says nothing about the situation where that standard is not supported by an implementation.

Other Languages

Few languages get involved in this level of detail.

Common Implementations

HP–was DEC– VAX has a bit representation for reserved, Cray a bit representation for indefinite. They behave like NaNs.

Coding Guidelines

If developers only make use of the functionality specified by the standard, this representation detail is unlikely to be of significance to them.

---

footnote 18

18) Evaluation of **FLT_ROUNDS** correctly reflects any execution-time change of rounding mode through the function **fesetround** in **<fenv.h>**. 359

Commentary

float.h 348 suitable for #if

This footnote clarifies the intended behavior for the FLT_ROUNDS (an earlier requirement also points out that this macro need not be a constant expression), at least for values between 0 and 3. It is possible for FLT_ROUNDS to have a value of -1 (or perhaps a value indicating an implementation-defined rounding behavior) and be unaffected by changes in rounding mode through calls to the function fesetround. The evaluation of FLT_ROUNDS also needs to correctly reflect any translation-time change of rounding mode through, for instance, use of a **#pragma** directive.

FLT_ROUNDS 352

Like errno FLT_ROUNDS provides a method of accessing information in the execution environment.

C90

Support for the header **<fenv.h>** is new in C99. The C90 Standard did not provide a mechanism for changing the rounding direction.

C++

Support for the header **<fenv.h>** and the fesetround function is new in C99 and is not specified in the C++ Standard.

---

footnote 19

19) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. 360

Commentary

complex types 500

Floating types are made up of the real and complex types

C90

Support for complex types is new in C99.

C++

The complex types are a template class in C++. The definitions of the instantiation of these classes do not specify that the evaluation format shall be the same as for the real types. But then, the C++ Standard does not specify the evaluation format for the real types.

Other Languages

The complex types in Fortran and Ada follow the same evaluation rules as the real types in those languages.

For example, if **FLT_EVAL_METHOD** is 1, then the product of two **float _Complex** operands is represented in the **double _Complex** format, and its parts are evaluated to **double**. 361

Commentary

This restatement of the specification makes doubly sure that there is no possible ambiguity of the intent.

C++

The C++ Standard does not specify a FLT_EVAL_METHOD mechanism.

---

1 evaluate operations and constants of type **float** and **double** to the range and precision of the **double** type, evaluate **long double** operations and constants to the range and precision of the **long double** type; 362

Commentary

Historically, this is the behavior many experienced C developers have come to expect.

Common Implementations

This is how translators are affected (i.e., existing source expects this behavior and a vendor wants their product to handle existing source) by the original K&R behavior.

Coding Guidelines

Although this behavior is implicitly assumed in much existing source, inexperienced developers may not be aware of it. As such it is an educational rather than a coding guidelines issue.

---

363 2 evaluate all operations and constants to the range and precision of the **long double** type.

Commentary

This choice reflects the fact that some hardware floating-point units (e.g., the Intel x86) support a single evaluation format for all floating-point operations. Continually having to convert the intermediate results, during the evaluation of an expression, to another format imposes an execution-time overhead that translator vendors may feel their customers will not tolerate.

Common Implementations

The Intel x86[627] performs floating-point operations in an 80-bit extended representation (later versions of this processor contained additional instructions that operated on a 32-bit representation only). Some translators, for this processor, use a **long double** type that has 96 bits in its object representation and 80 bits in its value representation.

574 object representation
595 value representation

---

364 All other negative values for **FLT_EVAL_METHOD** characterize implementation-defined behavior.

Commentary

The standard does not say anything about other possible positive values, not even reserving them for future revisions.

Common Implementations

One form of evaluation format seen in some implementations is widest-need.[52] This format was also described in the Floating-point C Extensions technical report[1019] produced by the NCEG (Numerical C Extensions Group) subcommittee of X3J11. Widest-need examines a full expression for the widest real type in that expression. All of the floating-point operations in that expression are then evaluated to that widest type.

0 X3J11

Coding Guidelines

The issues behind a widest-need evaluation strategy are the same as those for integer types. A change of type of one operand can affect the final result of an expression. Recommending the use of a single floating type is not such an obvious solution (as it is for integer types). Producing a correctly rounded result requires the use of two different floating types. This is a complex issue and your author knows of no simple guideline recommendation that might be applicable.

64 correctly rounded result

Example

```
1  extern float f1, f2;
2  extern double d1, d2;
3  extern long double ld1, ld2;
4
5  void f(void)
6  {
7  f1 = f1 * f2; /* Widest type is float. */
8  /*
9   * Widest type is double.
10  * If FLT_EVAL_METHOD is 0 f1 * f2 would be evaluated in float.
11  */
```

```
12  f1 = d1 + f1 * f2;
13  }
```

The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign: 365

Commentary

The values listed below are all integers. The descriptions imply an integer type, but this is not explicitly specified (other requirements apply to integer values). One likely use of the values listed here are as the size in an array definition.

float.h 348 suitable for #if

C90

In C90 the only expression that was required to be a constant expression was FLT_RADIX. It was explicitly stated that the others need not be constant expressions; however, in most implementations, the values were constant expressions.

C++

18.2.1p3 *For all members declared* `static const` *in the* `numeric_limits` *template, specializations shall define these values in such a way that they are usable as integral constant expressions.*

Other Languages

Java specifies the exact floating-point format to use, so many of the following values are already implicitly defined in that language. It also defines the classes `java.lang.Float` and `java.lang.Double` which contain a minimal set of related information.

Coding Guidelines

Like other identifiers, which are constant expressions, defined by the C Standard, they represent a symbolic property. As discussed elsewhere, making use of the property rather than a particular arithmetic value has many advantages.

symbolic name 822

FLT_RADIX — radix of exponent representation, b 366

FLT_RADIX **2**

Commentary

A poorly worded way of saying the radix of the significand. The value of this macro is an input parameter to many formulas used in the calculation of rounding errors. The constants used in some numerical algorithms may depend on the value of FLT_RADIX. For instance, minimizing the error in the calculation of sine requires using different constant values for different radixs.

The radix used by humans is 10. A fraction that has a finite representation in one radix may have an infinitely repeating representation in another radix. This only occurs when there is some prime number, p, that divides one radix but not the other. There is no prime divisor of 2, 8, or 16 that does not also divide 10. Therefore, it is always possible to exactly represent binary, octal, or hexadecimal fractions exactly as base 10 fractions. The prime number 5 divides 10, but not 2, 8, or 16. Therefore, there are decimal fractions that have no exact representation in binary, octal, or hexadecimal. For instance, 0.1_{10} has an infinitely recurring representation in base 2 (i.e., $1.10011001100110011\ldots_2 \times 2^{-4}$).

When FLT_RADIX has a value other than 2, the precision wobbles. For instance, with a FLT_RADIX of 16 for the type **float**, there are 4×6 or 24 bits of significand, but only 21 bits of precision that can be counted

on. The shifting of the bits in the significand, to ensure correct alignment, occurs in groups of 4, meaning that 3 bits can be 0. A large radix has the advantages of a large range (of representable values) and high speed (of execution). The price paid for this advantage may be less accuracy and larger errors in the results of arithmetic operations. A radix of 2 provides the best accuracy (it minimizes the root mean square round-off error[158]).

The base of the exponent affects the significand if it has a value other than 2. In these cases the significand will contain leading zeros for some floating-point values. A discussion of the trade-off involved in choosing the base and the exponent range is given by Richman.[1165]

C++

```
static const int radix;
```

18.2.1.2p16

For floating types, specifies the base or radix of the exponent representation (often 2).[185)]

Other Languages

A numeric inquiry function, RADIX, which returns the radix of a real or integer argument was added to Fortran 90. Few other languages get involved in exposing such details to developers.

Common Implementations

Because of the widespread use of IEC 60559 by processor vendors, the most commonly encountered value is 2. Values of 8 and 16 are still sometimes encountered, but such usage is becoming rarer as the machines supporting them are scrapped (or move to support IEC 60559, like the IBM 390, and existing software is migrated); although software emulation of floating-point operators sometimes uses a non-2 value.[267]

The Unisys A Series[1389] uses a FLT_RADIX of 8. The Motorola 68000 processor[968] supports a packed decimal real format that has a FLT_RADIX of 10. This processor also supports IEC 60559 format floating-point types. The original IBM 390 floating-point format uses a FLT_RADIX of 16. Later versions of this processor also support the IEC 60559 format.[1207] The Data General Nova[319] also uses a FLT_RADIX of 16. The Los Alamos MANIAC II used a radix of 256, only one was every built, using valves, in 1955. The Moscow State University SETUN experimental computer[1442] used a radix of 3.

The advantage of a larger radix is that it increases the probability[1323] that the exponents of two floating-point values, of a binary operator, will be the same, reducing the need to align the value significands. A larger radix is also likely to require less work to normalise result of an operation.

The advantage of a smaller radix is that it has a smaller maximum and average representation error[249] (for the same total number of bits in the floating-point representation).

A radix of 2 introduces a significant performance penalty for software implementations of floating-point operations and a radix of 8 has been found[267] to be a better choice.

Hardware support for a radix of 10 has started to take off in environments where accurate calculations involving values represented in *human form* is important[298] (Cowlishaw[297] provides a specification of IEC 60559 conforming decimal arithmetic). There is a strong possibility that use of radix 10 will eventually become the norm, as the economic (e.g., impact on chip fabrication costs) and technical (e.g., runtime efficiency) costs become less important than the benefits, to end-users, of decimal arithmetic.

Example

Malcolm[889] provides Fortran source that computes the radix used by the processor on which the code executes.

```
#include <float.h>

#if FLT_RADIX != 16
#error Go away!  This program only translates on old cranky hosts.
#endif
```

*_MANT_DIG macros — number of base-FLT_RADIX digits in the floating-point significand, p 367

FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG

Commentary

Assumed to be an integer (which it is for almost all implementations).

Rationale The term FLT_MANT_DIG stands for "float mantissa digits." The Standard now uses the more precise term *significand* rather than *mantissa*.

Other Languages

Few languages get involved in exposing such details to the developer. A numeric inquiry function, DIGITS, which returns the number of base b digits in the significand of the argument, was added in Fortran 90.

Common Implementations

The IEC 60559 Standard defines single-precision as 24 bits (base-2 digits) and double-precision as 53 bits. It also defines a single extended as having 32 or more bits and a double extended as having 64 or more bits. In the actual bit representation (for single and double) there is always 1 less bit. This is because normalized numbers contain an implied 1 in the most significant bit and subnormals an implied 0 in that position.

subnormal numbers 338

In many freestanding environments the host processor does not support floating-point operations in hardware; they are emulated via library calls. To reduce the significant execution-time overhead of performing floating-point operations in software, a 16-bit significand is sometimes used (the exponent and sign occupy 8 bits, giving a total of 24 bits).

Some processors (e.g., the ADSP–2106x[29] and Intel XScale microarchitecture[629]) support single extended by adding an extra byte to the type representation, giving a 40-bit significand. Some processors also support a floating-point format, in hardware, that cannot represent the full range of values supported by IEC 60559. For instance, the ADSP–2106x[29] has what it calls a *short word* floating-point format— occupying 16 bits with a 4-bit exponent, sign bit and an 11-bit significand.

Example

Malcolm[889] provides Fortran source that computes the number of mantissa digits used by the processor on which the code executes.

```
1  #include <float.h>
2
3  #if (DBL_MANT_DIG != 53) || (FLT_MANT_DIG != 24)
4  #error Please rewrite the hex floating-point literals
5  #endif
```

DECIMAL_DIG macro — number of decimal digits, n, such that any floating-point number in the widest supported floating type with p_{max} radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value, 368

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

DECIMAL_DIG 10

Commentary

The conversion process here is base-FLT_RADIX⇒base-10⇒base-FLT_RADIX (the opposite conversion ordering is discussed in the following C sentence). The binary fraction 0.00011001100110011001100110011 is exactly equal to the decimal fraction 0.09999999962747097015380859375, which might suggest that a DECIMAL_DIG value of 10 is not large enough. However, the second part of the above requirement is that the number be converted back without change, not that the exact decimal representation be used. If FLT_RADIX is less than 10, then for the same number of digits there will be more than one decimal representation for each binary number. The number N of radix-B digits required to represent an n-digit FLT_RADIX floating-point number is given by the condition (after substituting C Standard values):[904]

$$10^{N-1} > FLT\_RADIX^{LDBL\_MANT\_DIG} \tag{368.1}$$

Minimizing for N, we get:

$$N = 2 + \frac{LDBL\_MANT\_DIG}{\log_{FLT\_RADIX} 10} \tag{368.2}$$

When FLT_RADIX is 2, this simplifies to:

$$N = 2 + \frac{LDBL\_MANT\_DIG}{3.321928095} \tag{368.3}$$

By using fewer decimal digits, we are accepting that the decimal value may not be the one closest to the binary value. It is simply a member of the set of decimal values having the same binary value that is representable in DECIMAL_DIG digits. For instance, the decimal fraction 0.1 is closer to the preceding binary fraction than any other nearby binary fractions.[1287]

379 DECIMAL_DIG conversion recommended practice

When b is not a power of 10, this value will be larger than the equivalent *_DIG macro. But not all of the possible combinations of DECIMAL_DIG decimal digits can be generated by a conversion. The number of representable values between each power of the radix is fixed. However, each successive power of 10 supports a greater number of representable values (see Figure 368.1). Eventually the number of representable decimal values, in a range, is greater than the number of representable p radix values. The value of DECIMAL_DIG denotes the power of 10 just before this occurs.

335 precision floating-point

C90

Support for the DECIMAL_DIG macro is new in C99.

C++

Support for the DECIMAL_DIG macro is new in C99 and specified in the C++ Standard.

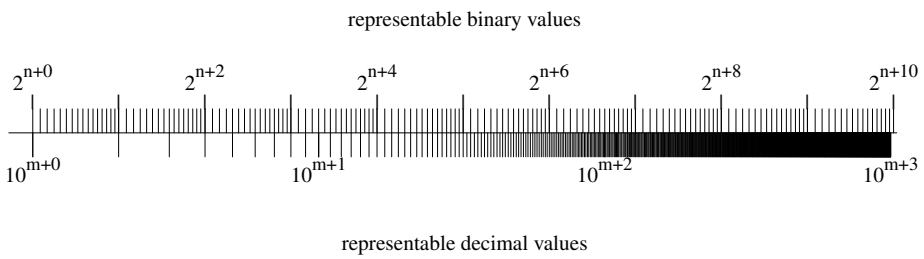


Figure 368.1: Representable powers of 10 and powers of 2 on the real line.

Other Languages

Few other languages get involved in exposing such details to the developer.

Common Implementations

A value of 17 would be required to support IEC 60559 double precision. A value of 9 is sufficient to support IEC 60559 single precision.

long double Apple

The format used by Apple on the PowerPC[52] to represent the **long double** type is the concatenation of two **double** types. Apple recommends that the difference in exponents, between the two doubles, be 54. However, it is not possible to guarantee this will always be the case, giving the representation an indefinite precision. The number of decimal digits needed to ensure that a cycle of conversions delivers the original value is proportional to the difference between the exponents in the two doubles. When the least significant double has a value of zero, the difference can be very large.

The following example is based on the one given in the Apple Macintosh PowerPC numerics documentation.[52] If an object with type **double**, having the value `1.2`, is assigned to an object having **long double** type, the least significant bits of the significand are given the zero value. In hexadecimal (and decimal to 34 digits of accuracy):

```
0x3FF33333 33333333 00000000 00000000
  1.19999999999999995559107901493738
```

The decimal form is the closest 34-digit approximation of the long double number (represented using double-double). It is also the closest 34-decimal digit approximation to an infinitely precise binary value whose exponent is 0 and whose fractional part is represented by 13 sequences of `0011` followed by 52 binary zeros, followed by some nonzero bits. Converting this decimal representation back to a binary representation, the Apple PowerPC Numerics library returns the closest double-double approximation of the infinitely precise value, using all of the bits of precision available to it. It will use all 53 bits in the head and 53 bits in the tail to store nonzero values and adjust the exponent of the tail accordingly. The result is:

```
0x3FF33333 33333333 xxxyzzzz zzzzzzzz
```

where xxx represents the sign and exponent of the tail, and `yzzz...` represents the start of a nonzero value. Because the tail is always nonzero, this value is guaranteed to be not equal to the original value.

Implementations add additional bits to the exponent and significand to support a greater range of values and precision, and most keep the bits representing the various components contiguous. The Unisys A Series[1389] represents the type **double** using the same representation as type **float** in the first word, and by having additional exponent and significand bits in a second word. The external effect is the same. But it is an example of how developer assumptions about representation, in this case bits being contiguous, can be proved wrong.

Coding Guidelines

One use of this macro is in calculating the amount of storage needed to hold the character representation, in decimal, of a floating-point value. The definition of the macro excludes any characters (digits or otherwise) that may be used to represent the exponent in any printed representation.

The only portable method of transferring data having a floating-point type is to use a character-based representation (e.g., a list of decimal floating-point numbers in character form). For a given implementation, this macro gives the minimum number of digits that must be written if that value is to be read back in without change of value.

```
printf("%#.*g", DECIMAL_DIG, float_valu);
```

Space can be saved by writing out fewer than `DECIMAL_DIG` digits, provided the floating-point value contains less precision than the widest supported floating type. Trailing zeros may or may not be important; the issues involved are discussed elsewhere.

Converting a floating-point number to a decimal value containing more than DECIMAL_DIG digits may, or may not, be meaningful. The implementation of the printf function may, or may not, choose to convert to the decimal value closest to the internally represented floating-point value, while other implementations simply produce garbage digits.[1287]

Example

```
#include <float.h>

/*
 * Array big enough to hold any decimal representation (at least for one
 * implementation).  Extra characters needed for sign, decimal point,
 * and exponent (which could be six, E-1234, or perhaps even more).
 */
#if LDBL_MAX_10_EXP < 10000

char float_digits[DECIMAL_DIG + 1 + 1 + 6 + 1];

#else
#error Looks like we need to handle this case
#endif
```

---

369 — number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits, * _DIG macros

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

FLT_DIG	**6**
DBL_DIG	**10**
LDBL_DIG	**10**

Commentary

The conversion process here is base-10⇒base-FLT_RADIX⇒base-10 (the opposite ordering is described elsewhere). These macros represent the maximum number of decimal digits, such that each possible digit sequence (value) maps to a different (it need not be exact) radix b representation (value). If more than one decimal digit sequence maps to the same radix b representation, it is possible for a different decimal sequence (value) to be generated when the radix b form is converted back to its decimal representation.

368 DECIMAL_DIG macro

C++

18.2.1.2p9

```
static const int digits10;
```

Number of base 10 digits that can be represented without change.

Footnote 184

Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

The Fortran 90 `PRECISION` inquiry function is defined as `INT((p-1) * LOG10(b)) + k`, where `k` is 1 is `b` is an integer power of 10 and 0 otherwise.

Example

```
1  #include <limits.h>
2  #include <stdio.h>
3
4  float real_float;
5
6  int main(void)
7  {
8  for (int findex = INT_MAX; findex > INT_MAX-20; findex--)
9     {
10    real_float=(float)findex; /* Just in case we have a 'clever' optimizer. */
11    printf("I=%d, F1=%f, F2=%f\n", findex, (float)findex, real_float);
12    }
13 }
```

---

*_MIN_EXP — minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, e_{min} 370

FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP

Commentary

*_MIN macros 378 These values are essentially the minimum value of the exponent used in the internal floating-point representation. The `*_MIN` macros provide constant values for the respective minimum normalized floating-point value. No minimum values are given in the standard. The possible values can be calculated from the following:

$$FLT\_MIN\_EXP = \frac{FLT\_MIN\_10\_EXP}{\log(FLT\_RADIX)} \pm 1 \tag{370.1}$$

C++

18.2.1.2p23

```
static const int min_exponent;
```

Minimum negative integer such that `radix` *raised to that power is in the range of normalised floating point numbers.*[189)]

Footnote 189 *Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.*

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

Fortran 90 contains the inquiry function MINEXPONENT which performs a similar function.

Common Implementations

In IEC 60559 the value for single-precision is -125 and for double-precision -1021. The two missing values (available in the biased notation used to represent the exponent) are used to represent 0.0, subnormals, infinities, and NaNs.

338 floating types can represent

Coding Guidelines

The usage of these macros in existing code is so rare that reliable information on incorrect usage is not available, making it impossible to provide any guideline recommendations. (The rare usage could also imply that a guideline recommendation would not be worthwhile).

---

371 — minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{min}-1} \rceil$ *_MIN_10_EXP

FLT_MIN_10_EXP	**-37**
DBL_MIN_10_EXP	**-37**
LDBL_MIN_10_EXP	**-37**

Commentary

Making this information available as an integer constant allows it to be accessed in a **#if** preprocessing directive.

These are the exponent values for normalized numbers. If subnormal numbers are supported, the smallest representable value is likely to have an exponent whose value is FLT_DIG, DBL_DIG, and LDBL_DIG less than (toward negative infinity) these values, respectively.

338 subnormal numbers

The Committee is being very conservative in specifying the minimum values for the exponents of the types **double** and **long double**. An implementation is permitted to define the same range of exponents for all floating-point types. There may be normalized numbers whose respective exponent value is smaller than the values given for these macros; for instance, the exponents appearing in the *_MIN macros. The power of 10 exponent values given for these *_MIN_10_EXP macros can be applied to any normalized significand.

C++

```
static const int min_exponent10;
```

18.2.1.2p25

Minimum negative integer such that 10 raised to that power is in the range of normalised floating point numbers.[190)]

Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP. Footnote 190

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2p4

Common Implementations

The value of DBL_MIN_10_EXP is usually the same as FLT_MIN_10_EXP or LDBL_MIN_10_EXP. In the latter case a value of -307 is often seen.

*_MAX_EXP — maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, e_{max} 372

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

Commentary

*_MIN_EXP 370 FLT_RADIX to the power *_MAX_EXP is the smallest large number that cannot be represented (because of limited exponent range).

C++

18.2.1.2p27

```
static const int max_exponent;
```

Maximum positive integer such that `radix` *raised to the power one less than that integer is a representable finite floating point number.*[191)]

Footnote 191 *Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.*

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

Fortran 90 contains the inquiry function MAXEXPONENT which performs a similar function.

Common Implementations

In IEC 60559 the value for single-precision is 128 and for double-precision 1024.

*_MAX_10_EXP — maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p})b^{e_{max}}) \rfloor$ 373

FLT_MAX_10_EXP	**+37**
DBL_MAX_10_EXP	**+37**
LDBL_MAX_10_EXP	**+37**

Commentary

*_MIN_10_EXP 371 As in choosing the *_MIN_10_EXP values, the Committee is being conservative.

C++

18.2.1.2p29

```
static const int max_exponent10;
```

Maximum positive integer such that 10 raised to that power is in the range of normalised floating point numbers.

Footnote 192

Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2

Common Implementations

The value of DBL_MAX_10_EXP is usually the same as FLT_MAX_10_EXP or LDBL_MAX_10_EXP. In the latter case a value of 307 is often seen.

---

374 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown: floating values listed

Commentary

This is a requirement on the implementation. The requirement that they be constant expressions ensures that they can be used to initialize an object having static storage duration.

The values listed represent a floating-point number. Their equivalents in the integer domain are required to have appropriate promoted types. There is no such requirement specified for these floating-point values. 822 symbolic name 303 integer types sizes

C90

C90 did not contain the requirement that the values be constant expressions.

C++

This requirement is not specified in the C++ Standard, which refers to the C90 Standard by reference.

---

375 — maximum representable finite floating-point number, $(1 - b^{-p})b^{e_{max}}$

FLT_MAX	**1E+37**
DBL_MAX	**1E+37**
LDBL_MAX	**1E+37**

Commentary

There is no requirement that the type of the value of these macros match the real type whose maximum they denote. Although some implementations include a representation for infinity, the definition of these macros require the value to be finite. These values correspond to a FLT_RADIX value of 10 and the exponent values given by the *_MAX_10_EXP macros. 373 *_MAX_10_EXP

The HUGE_VAL macro value may compare larger than any of these values.

C++

```
static T max() throw();
```

Maximum finite value.[182] 18.2.1.2p4

Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc. Footnote 182

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

The class `java.lang.Float` contains the member:

```
1  public static final float MAX_VALUE = 3.4028235e+38f
```

The class `java.lang.Double` contains the member:

```
1  public static final double MAX_VALUE = 1.7976931348623157e+308
```

Fortran 90 contains the inquiry function `HUGE` which performs a similar function.

Common Implementations

Many implementations use a suffix to give the value a type corresponding to what the macro represents. The IEC 60559 values of these macros are:

```
single float FLT_MAX        3.40282347e+38
double float DBL_MAX        1.7976931348623157e+308
```

EXAMPLE 380 minimum floating-point representation

EXAMPLE 381 IEC 60559 floating-point

Coding Guidelines

How many calculations ever produce a value that is anywhere near `FLT_MAX`? The known Universe is thought to be 3×10^{29}mm in diameter, 5×10^{19} milliseconds old, and contain 10^{79} atoms, while the Earth is known to have a mass of 6×10^{24}Kg.

Floating-point values whose magnitude approaches `DBL_MAX`, or even `FLT_MAX` are only likely to occur as the intermediate results of calculating a final value. Very small numbers are easily created from values that do not quite cancel. Dividing by a very small value can lead to a very large value. Very large values are thus more often a symptom of a problem, rounding errors or poor handling of values that almost cancel, than of an application meaningful value.

Some processors saturate to the maximum representable value on overflow, while others return an infinity. Testing whether an operation will overflow is one use for these identifiers. However, in C99, tests such as (`x > LDBL_MAX`) can now be replaced by the `isinf` macro.

Example

```
1  #include <float.h>
2
3  #define FALSE 0
4  #define TRUE 1
5
6  extern float f_glob;
7
8  _Bool f(float p1, float p2)
9  {
10 if (f_glob > (FLT_MAX / p1))
11    return FALSE;
12
13 f_glob *= p1;
14
15 if (f_glob > (FLT_MAX - p2))
16    return FALSE;
17
18 f_glob += p2;
19
20 return TRUE;
21 }
```

376 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

Commentary

The previous discussion is applicable here.

374 floating values listed

377 — the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p} *_EPSILON

FLT_EPSILON	**1E-5**
DBL_EPSILON	**1E-9**
LDBL_EPSILON	**1E-9**

Commentary

The Committee is being very conservative in specifying these values. Although IEC 60559 arithmetic is in common use, there are several major floating-point implementations of it that do not support an extended precision. The Committee could not confidently expect implementations to support the type **long double** containing greater accuracy than the type **double**.

29 IEC 60559

Like the *_DIG macros more significand digits are required for the types **double** and **long double**.

369 *_DIG macros

C++

18.2.1.2p20

```
static T epsilon() throw();
```

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[187)]

Footnote 187

Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

Fortran 90 contains the inquiry function EPSILON, which performs a similar function.

Common Implementations

Some implementations (e.g., Apple) use a contiguous pair of objects having type **double** to represent an object having type **long double**. Such a representation creates a second meaning for LDBL_EPSILON. This is because, in such a representation, the least value greater than 1.0 is 1.0+LDBL_MIN, a difference of LDBL_MIN (which is not the same as $b^{(1-p)}$)— the correct definition of *_EPSILON. Their IEC 60559 values are:

368 long double Apple

```
FLT_EPSILON     1.19209290e-7 /* 0x1p-23 */
DBL_EPSILON     2.2204460492503131e-16 /* 0x1p-52 */
```

Coding Guidelines

It is a common mistake for these values to be naively used in equality comparisons:

```
#define EQUAL_DBL(x, y) ((((x)-DBL_EPSILON) < (y)) && \
                             (((x)+DBL_EPSILON) > (y)))
```

This test will only work as expected when x is close to 1.0. The difference value not only needs to scale with x, (x + x*DBL_EPSILON), but the value DBL_EPSILON is probably too small (equality within 1 ULP is a very tight bound):

ULP 346

```
#define EQUAL_DBL(x, y) ((((x)*(1.0-MY_EPSILON)) < (y)) && \
                             (((x)*(1.0+MY_EPSILON)) > (y)))
```

Even this test fails to work as expected if x and y are subnormal values. For instance, if x is the smallest subnormal and y is just 1 ULP bigger, y is twice x.

Another, less computationally intensive, method is to subtract the values and check whether the result is within some scaled approximation of zero.

```
#include <math.h>

_Bool equalish(double f_1, double f_2)
{
int exponent;
frexp(((fabs(f_1) > fabs(f_2)) ? f_1 : f_2), &exponent);
return (fabs(f_1-f_2) < ldexp(MY_EPSILON, exponent));
}
```

---

*_MIN macros

— minimum normalized positive floating-point number, $b^{e_{min}-1}$

378

FLT_MIN	**1E-37**
DBL_MIN	**1E-37**
LDBL_MIN	**1E-37**

Commentary

These values correspond to a FLT_RADIX value of 10 and the exponent values given by the *_MIN_10_EXP macros. There is no requirement that the type of these macros match the real type whose minimum they denote. Implementations that support subnormal numbers will be able to represent smaller quantities than these.

*_MIN_10_EXP 371
subnormal numbers 338

C++

18.2.1.2p1

> *static T min() throw();*
>
> *Maximum finite value.*[181)]

Footnote 181

> *Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.*

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

The class `java.lang.Float` contains the member:

```
1  public static final float MIN_VALUE = 1.4e-45f;
```

The class `java.lang.Double` contains the member:

```
1  public static final double MIN_VALUE = 5e-324;
```

which are the smallest subnormal, rather than normal, values.

Fortran 90 contains the inquiry function `TINY` which performs a similar function.

Common Implementations

Their IEC 60559 values are:

```
FLT_MIN          1.17549435e-38f
DBL_MIN          2.2250738585072014e-308
```

Implementations without hardware support for floating point sometimes chose the minimum required limits because of the execution-time overhead in supporting additional bits in the floating-point representation.

Coding Guidelines

How many calculations ever produce a value that is anywhere near as small as `FLT_MIN`? The hydrogen atom weighs 10^{-26}Kg and has an approximate radius of $5{\times}10^{-11}$ meters, well within limits. But current theories on the origin of the Universe start at approximately 10^{-36} seconds, a very small number. However, writers of third-party libraries might not know whether their users are simulating the Big Bang, or weighing groceries. They need to ensure that all cases are handled.

Given everyday physical measurements, which don't have very small values, where can very small numbers originate? Subtracting two floating-point quantities that differ by 1 ULP, for instance, produces a value that is approximately 10^{-5} smaller. Such a difference can result from random fluctuations in the values input to a program, or because of rounding errors in calculations. Producing a value that is close to `FLT_MIN` invariably requires either a very complex calculation, or an iterative algorithm using values from previous iterations. Intermediate results that are expected to produce a value of zero may in fact deliver a very small value. Subsequent tests against zero fail and the very small value is passed through into further calculations. One solution to this problem is to have a relatively wide test of zeroness. In many physical systems a value that is a factor of 10^{-6} smaller than the smallest measurable quantity would be considered to be zero.

Rev 378.1 Floating-point comparisons against zero shall take into account the physical properties or engineering tolerances of the system being controlled or simulated.

There might be some uncertainty in the interpretation of the test `(abs(x) < FLT_MIN);` is it an approximate test against zero, or a test for a subnormal value? The C Standard now includes the `fpclassify` macro for obtaining the classification of its argument, including subnormal.

Example

```
1  #include <math.h>
2
3  #define MIN_TOLERANCE (1e-9)
4
5  _Bool inline effectively_zero(float valu)
6  {
7  return (abs(valu) < MIN_TOLERANCE);
8  }
```

Recommended practice

DECIMAL_DIG conversion recommended practice

Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back should be the identity function. 379

Commentary

Why is this a recommended practice? Unfortunately many existing implementations of `printf` and `scanf` do a poor job of base conversions, and they are not the identity functions.

To claim conformance to both C99 and IEC 60559 (Annex F in force), the requirements of F.5 Binary-decimal conversion must be met. Just making use of IEC 60559 floating-point hardware is not sufficient. The I/O library can still be implemented incorrectly and the conversions be wrong.

Rationale When the radix b is not a power of 10, it can be difficult to find a case where a decimal number with $p \times \log_{10} b$ digits fails. Consider a four-bit mantissa system (that is, base $b = 2$ and precision $p = 4$) used to represent one-digit decimal numbers. While four bits are enough to represent one-digit numbers, they are not enough to support the conversions of decimal to binary and back to decimal in all cases (but they are enough for most cases). Consider a power of 2 that is just under 9.5e21, for example, 2^{73} = 9.44e21. For this number, the three consecutive one-digit numbers near that special value and their round-to-nearest representations are:

```
  9e21    1e22    2e22
0xFp69  0x8p70  0x8p71
```

No problems so far; but when these representations are converted back to decimal, the values as three-digit numbers and the rounded one-digit numbers are:

```
8.85e21  9.44e21  1.89e22
   9e21     9e21     2e22
```

and we end up with two values the same. For this reason, four-bit mantissas are not enough to start with any one-digit decimal number, convert it to a binary floating-point representation, and then convert back to the same one-digit decimal number in all cases; and so p radix b digits are (just barely) not enough to allow any decimal numbers with $p \times \log_{10} b$ digits to do the round-trip conversion. p radix b digits are enough, however, for $(p-1) \times \log_{10} b$ digits in all cases.

The issues involved in performing correctly rounded decimal-to-binary and binary-to-decimal conversions are discussed mathematically by Gay.[475]

C90

The *Recommended practice* clauses are new in the C99 Standard.

C++

There is no such macro, or requirement specified in the C++ Standard.

Other Languages

The specification of the Java base conversions is poor.

Common Implementations

Experience with testing various translators shows that the majority don't, at the time of this publication, implement this Recommended Practice. The extent to which vendors will improve their implementations is unknown.

There is a publicly available set of tests for testing binary to decimal conversions.[1380]

Coding Guidelines

A Recommended Practice shall not be relied on to be followed by an implementation.

---

380 EXAMPLE 1 The following describes an artificial floating-point representation that meets the minimum requirements of this International Standard, and the appropriate values in a **<float.h>** header for type **float**:

EXAMPLE minimum floating-point representation

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
FLT_RADIX                       16
FLT_MANT_DIG                     6
FLT_EPSILON         9.53674316E-07F
FLT_DIG                          6
FLT_MIN_EXP                    -31
FLT_MIN             2.93873588E-39F
FLT_MIN_10_EXP                 -38
FLT_MAX_EXP                    +32
FLT_MAX             3.40282347E+38F
FLT_MAX_10_EXP                 +38
```

Commentary

Note that this example has a FLT_RADIX of 16, not 2.

---

381 EXAMPLE 2 The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in IEC 60559,[20] and the appropriate values in a **<float.h>** header for types **float** and **double**:

EXAMPLE IEC 60559 floating-point

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                       2
DECIMAL_DIG                    17
FLT_MANT_DIG                   24
FLT_EPSILON         1.19209290E-07F // decimal constant
FLT_EPSILON               0X1P-23F // hex constant
FLT_DIG                         6
FLT_MIN_EXP                  -125
FLT_MIN             1.17549435E-38F // decimal constant
FLT_MIN                   0X1P-126F // hex constant
FLT_MIN_10_EXP                -37
FLT_MAX_EXP                  +128
FLT_MAX             3.40282347E+38F // decimal constant
FLT_MAX             0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                +38
DBL_MANT_DIG                  53
DBL_EPSILON 2.2204460492503131E-16 // decimal constant
DBL_EPSILON                0X1P-52 // hex constant
DBL_DIG                       15
DBL_MIN_EXP                -1021
DBL_MIN    2.2250738585072014E-308 // decimal constant
DBL_MIN                  0X1P-1022 // hex constant
DBL_MIN_10_EXP                -307
DBL_MAX_EXP                  +1024
```

```
DBL_MAX    1.7976931348623157E+308 // decimal constant
DBL_MAX     0X1.fffffffffffffP1023 // hex constant

DBL_MAX_10_EXP                +308
```

If a type wider than **double** were supported, then **DECIMAL_DIG** would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then **DECIMAL_DIG** would be 21.

Commentary

The values given here are important in that they are the most likely values to be provided by a conforming implementation using IEC 60559, which is what the majority of modern implementations use. These values correspond to the IEC 60559 single- and double-precision formats. This standard also defines extended single and extended double formats, which contain more bits in the significand and greater range in the exponent.

Note that this example gives the decimal and hexadecimal floating-constant representation for some of the macro definitions. A real header will only contain one of these definitions.

C90

The C90 wording referred to the ANSI/IEEE-754–1985 standard.

---

footnote 20

20) The floating-point model in that standard sums powers of b from zero, so the values of the exponent limits are one less than shown here. 382

Commentary

Fortran counts from 1, not 0 and the much of the contents of **<float.h>**, in C90, came from Fortran.

Forward references: conditional inclusion (6.10.1), complex arithmetic **<complex.h>** (7.3), extended multibyte and wide character utilities **<wchar.h>** (7.24), floating-point environment **<fenv.h>** (7.6), general utilities **<stdlib.h>** (7.20), input/output **<stdio.h>** (7.19), mathematics **<math.h>** (7.12). 383

6. Language

6.1 Notation

In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. 384

Commentary

A *terminal* is a token that can appear in the source code. A *nonterminal* is the name of a syntax rule used to group together zero or more terminals and other nonterminals. The nonterminals can be viewed as a tree. The root is the nonterminal *translation-unit*. The terminals are the leaves of this tree.

Syntax analysis is the processing of a sequence of terminals (as written in the source) via various nonterminals until the nonterminal *translation-unit* is reached. Failure to reach this final nonterminal, or encountering an unexpected sequence of tokens, is a violation of syntax.

The syntax notation used in the C Standard is not overly formal; it is often supported by text in the semantics clause. The C syntax can be written in LALR(1) form. (Although some reorganization of the productions listed in the standard is needed), assuming the **typedef** issue is fudged (the only way to know whether an identifier is a typedef name or not is to look it up in a symbol table, which introduces a context dependency; the alternative of syntactically treating a typedef name as an identifier requires more than one token lookahead.) This also happens to be the class of grammars that can be processed by `yacc` and many other parser generators.

```
A(B) /* Declare B to have type A, or call function A with argument B? */
```

The syntax specified in the C Standard effectively describes four different grammars:

1. A grammar whose start symbol is *preprocessing-token*; the input stream processed by this grammar contains the source characters output by translation phase 2.
2. A grammar whose start symbol is *preprocessing-file*; the input stream processed by this grammar contains the *preprocessing-token*s output by translation phase 3.
3. A grammar whose start symbol is *token*; the input to this grammar is a single *preprocessing-token*. The syntax of the characters forming the *preprocessing-token* need to form a valid parse of the *token* syntax.
4. A grammar whose start symbol is *translation-unit*; the input stream processed by this grammar contains the *token*s output by translation phase 6.

770 preprocessing token syntax
118 translation phase 2
1854 preprocessor directives syntax
124 translation phase 3
770 token syntax
1810 translation unit syntax
135 translation phase 6

The *preprocessor-token* and *token* syntax is sometimes known as the *lexical grammar* of C.

There are many factors that affect the decision of whether to specify language constructs using syntax or English prose in a Constraints clause. The C Standard took the approach of having a relatively simple, general syntax specification and using wording in constraints clauses to handle the special cases. There are techniques available (e.g., two-level grammars) for specifying the requirements (including the type rules) through syntax. This approach was famously taken, because of its impenetrability, in the specification of Algol 68[1407]

The first published standard for syntax notation was BS 6154:1981, *Method of Defining— Syntactic metalanguage*. Although a British rather than an International standard this document was widely circulated. It was also used as the base document for ISO/IEC 14977,[648] which specified an extended BNF. Most compiler books limit there discussion to LR and LL related methods. For a good introduction to a variety of parsing methods see Grune and Jacobs,[525] which is now freely available online.

C++

> *In the syntax notation used in this International Standard, syntactic categories are indicated by italic type, and literal words and characters in* `constant width` *type.* 1.6p1

The C++ grammar contains significantly more syntactic ambiguities than C. Some implementations have used mainly syntactic approaches to resolving these,[1467] while others make use of semantic information to guide the parse.[485] For instance, knowing what an identifier has been declared as, simplifies the parsing of the following:

```
template_name     < a , b > - 5  // equivalent to (template_name < a , b >) - 5)
non_template_name < a , b > - 5  // equivalent to (non_template_name < a) , (b > - 5)
```

Other Languages

Most computer languages definitions use some form of semiformal notation to define their syntax. The availability of parser generators is an incentive to try to ensure that the syntax is, or can be, rewritten in LALR(1) form.

Some languages are line-based, for instance Fortran (prior to Fortran 90). Each statement or declaration has its syntax and sequences of them can be used to build functions; there is no nesting of constructs over multiple statements.

The method used to analyze the syntax of a language can be influenced by several factors. Many Pascal translators used a handwritten recursive descent parser; the original, widely available, implementation of the language uses just this technique. Many Fortran translators use ad hoc techniques to process each statement or declaration as it is encountered, the full power of a parser generator not being necessary (also, it is easier to perform error recovery in an ad hoc approach).

Common Implementations

Most implementations use an automated tool-based approach to analyzing some aspects of the C syntax. Although tools do exist for automating the lexical grammar (e.g., `lex`) a handwritten lexer offers greater flexibility for handling erroneous input. The tool almost universally used to handle the language syntax is `yacc`, or one of its derivatives. The preprocessor syntax is usually handled in a handwritten ad hoc manner, although a few implementations use an automated tool approach.

Tools that do not require the input to be consolidated into tokens, but parsed at the character sequence level, are available.[1424]

Coding Guidelines

It is unlikely that a coding guideline recommendation would specify language syntax unless an extension was being discussed. Coding guideline documents that place restrictions on the syntax that can be used are rare.

A colon (*:*) following a nonterminal introduces its definition. 385

Commentary

This is a simple notation. Some syntax notations have been known to be almost as complex as the languages they describe.

C++

The C++ Standard does not go into this level of detail (although it does use this notation).

Other Languages

The notation *::=* is used in some language definitions and more formal specifications. The notation = is specified by the ISO Standard for extended BNF.[648]

Alternative definitions are listed on separate lines, except when prefaced by the words "one of". 386

Commentary

The Syntax clauses in the standard are written in an informal notation. As typified by this simple, nonformal rule.

Other Languages

The character | often used to indicate alternative definitions. This character is used by ISO/IEC 14977.

An optional symbol is indicated by the subscript "opt", so that 387

{ $expression_{opt}$ **}**

indicates an optional expression enclosed in braces.

Commentary

This is a more informal notation. In this example *expression* is optional; the braces are not optional.

Other Languages

Some languages use a more formal syntax notation where an empty alternative indicates that it is possible for a nonterminal to match against nothing (i.e., the symbol is optional).

When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens. 388

Commentary

There are some places in the main text where words are separated by hyphens instead of spaces.

C90

This convention was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not explicitly specify the conventions used. However, based on the examples given in clause 1.6 and usage within the standard, the conventions used appear to be the reverse of those used in C (i.e., syntactic categories are italicized and words are separated by hyphens).

Coding Guidelines

Companies may have typographical conventions for their documents which differ from those used by ISO. The issue of which typographical conventions to use in a company's coding guideline document is outside the scope of these coding guidelines.

---

389 A summary of the language syntax is given in annex A.

C90

The summary appeared in Annex B of the C90 Standard, and this fact was not pointed out in the normative text.

6.2 Concepts

Commentary

The concepts introduced here are identifiers (and their associated attributes), objects, and types.

6.2.1 Scopes of identifiers

---

390 An identifier can denote an object;

scope of identifiers

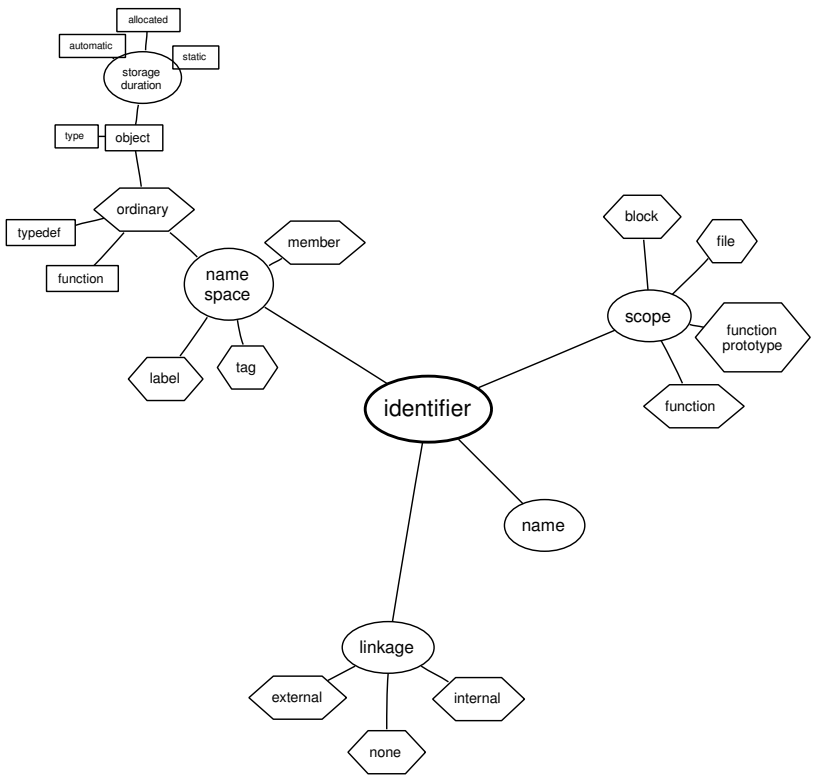


Figure 389.1: Attributes a C language identifier can have.

Commentary

This association is created by an explicit declaration.

Some objects do not have names— they are anonymous. An anonymous object can be created by a call to a memory-allocation function; an unnamed bit-field can be denoted by the identifier defined to have its containing union type.

Other Languages

Many languages use the term *variable* to denote what the C Standard calls an *object*. All languages provide a mechanism for declaring identifiers to denote objects (or variables).

Coding Guidelines

The use to which an identifier is put has been known to influence the choice of its name. This issue is fully discussed elsewhere.

naming conventions 792

---

a function; 391

Commentary

It is not possible to define an anonymous function within a strictly conforming program. Machine code can be copied into an object and executed on some implementations. Such copying and execution does not create something that is normally thought of as a function.

Coding Guidelines

Various coding guideline documents have recommended that names based on various parts of speech be used in the creating of identifier names denoting function. These recommendations are invariably biased towards the parts of speech found in English. (Not speaking any other human language, your author has not read documents written in other languages but suspects they are also biased toward their respective languages.) The general issue of identifier names is discussed elsewhere.

identifier introduction 792

---

a tag or a member of a structure, union, or enumeration; 392

Commentary

It is possible to create anonymous structure and union types, and anonymous members within these types. It is not possible to create an anonymous member of an enumeration. Even if there are gaps in the values assigned to each enumeration constant, there are no implicit named members.

C++

The C++ Standard does not define the term *tag*. It uses the terms *enum-name* (7.2p1) for enumeration definitions and *class-name* (9p1) for classes.

Common Implementations

Some translators support anonymous structure and union members within a structure definition; there was even a C9X revision proposal, WG14/N498 (submitted by one Ken Thompson):

```
1  struct S {
2            int mem1;
3            union {
4                   long umem1;
5                   float umem2;
6                  }; /* No member name. */
7           } s_o;
8
9  void f(void)
10 {
11 s_o.umem1=99; /* Translator deduces 'expected' member. */
12 }
```

There are several implementations that support this form of declaration. It is also supported in C++.

393 a typedef name;

Commentary

Character sequences such as **char**, **short** and **int** are types, not typedef names. They are also language keywords— a different lexical category from identifiers.

Common Implementations

Typedef names are specified as being identifiers, not keywords. However, most implementations treat them as a different syntactic terminal from identifiers. Translators usually performs a symbol table lookup just before translation phase 7, to see if an identifier is currently defined as a typedef name. This differentiation is made to simplify the parsing of C source and not visible to the user of a translator.

136 translation phase 7

Coding Guidelines

The issue of naming conventions typedef names is discussed elsewhere.

792 typedef naming conventions

394 a label name;

label name

Commentary

A label name occurs in a different syntactic context to other kinds of identifiers, which is how a translator deduces it is a label.

440 label name space
1722 labeled statements syntax

Other Languages

Fortran and Pascal require that label names consist of digits only.

Coding Guidelines

The issue of naming conventions for label names is discussed elsewhere.

792 label naming conventions

395 a macro name;

Commentary

An identifier can only exist as a macro name during translation phases 3 and 4.

1974 macro definition lasts until
124 translation phase 3
129 translation phase 4

Coding Guidelines

The issue of typedef names is discussed elsewhere.

792 macro naming conventions

396 or a macro parameter.

identifier macro parameter

Commentary

An identifier can only exist as a macro parameter during translation phases 3 and 4.

Function parameters are objects, not a special kind of identifier.

124 translation phase 3
129 translation phase 4

C++

The C++ Standard does not list macro parameters as one of the entities that can be denoted by an identifier.

71 parameter

Coding Guidelines

The issue of typedef names is discussed elsewhere.

792 typedef naming conventions

397 The same identifier can denote different entities at different points in the program.

identifier denote different entities

Commentary

The word *entities* is just a way of naming the different kinds of things that an identifier can refer to in C. The concept of scope allows the same identifier to be defined in a different scope, in the same name space, to refer to another entity. The concept of name space allows the same identifier to be defined in a different name space, in the same scope, to refer to another entity.

400 scope
438 name space

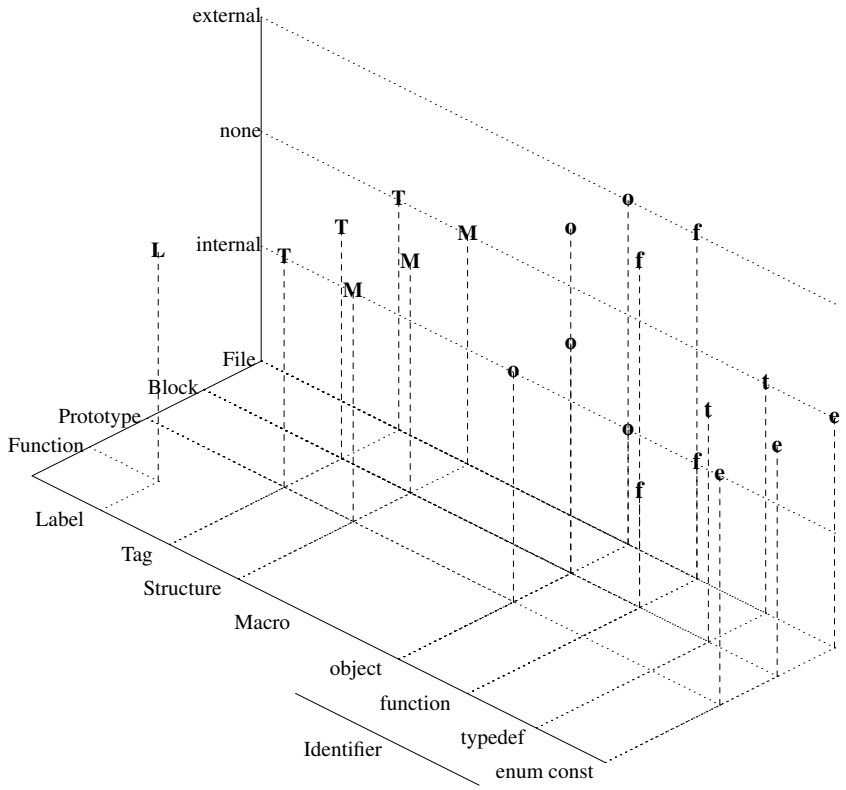


Figure 397.1: All combinations of linkage, scope, and name space that all possible kinds of identifiers, supported by C, can have. **M** refers to the members of a structure or union type. There is a separate name space for macro names and they have *no linkage*, but their scope has no formally specified name.

C++

The C++ Standard does not explicitly state this possibility, although it does include wording (e.g., 3.3p4) that implies it is possible.

Other Languages

Languages that support some kind of scoping rules usually allow the same identifier to denote different entities at different points in a program.

Cobol has a single scope and requires that identifiers be defined in a declaration section. This severely limits the extent to which the same identifier can be used to denote different entities. However, members of the structure data type can use the same names in different types. Some dialects of Basic have a single scope.

Coding Guidelines

Some of the costs and benefits of using the same identifier to denote different entities include:

- costs: an increase in the number of developer miscomprehensions when reading the source. For instance, the same identifier defined as a file scope, static linkage, object in one source file and a file scope, extern linkage, object in another source file could mislead the unwary developer, who was not aware of the two uses of the same name, into making incorrect assumptions about an assignment to one of the objects. The confusability of one identifier with another identifier is one of the major issues of identifier naming and is discussed elsewhere,

identifier 792 confusability

- benefits: an increase in reader recall performance through consistent use of identifier names to denote the same set of semantic attributes. For instance, having the same identifier denoting a member in

identifier 792 semantic associations

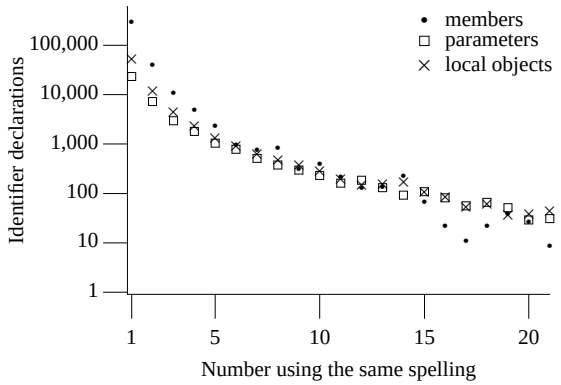


Figure 397.2: Number of declarations of an identifier with the same spelling in the same translation unit. Based on the translated form of this book's benchmark programs. Note that members of the same type are likely to be counted more than once (i.e., they are counted in every translation unit that declares them), while parameters and objects declared within function definitions are likely to be only counted once.

different structures can indicate that they all have the same usage (e.g., `x_cord` to hold the position on the x axis in a graph drawing program).

792 member naming conventions

The issues involved in making cost/benefit trade-offs related to identifier spelling and the role played by an identifier and are discussed elsewhere.

792 identifier syntax
1352 object role

Example

```
#define SAME(SAME) SAME /* Macro and macro parameter name spaces. */
/*
 * Additional lack of clarity could be obtained by replacing
 * any of the following identifiers, say id, by SAME(id).
 */

typedef struct SAME {          /* Tag name space. */
                    int SAME;  /* Unique name space of this structure definition. */
                    } SAME;    /* Ordinary identifier name space. */

SAME *(f(SAME SAME))(struct SAME SAME) /* Different scopes. */
{
SAME(SAME);                    /* A use (of the macro). */
if (SAME.SAME == sizeof(SAME)) /* More uses. */
   goto SAME;                  /* Label name space. */
else
   {                           /* A new scope. */
   enum SAME {
               SAME            /* Different name space. */
             } loc = SAME;     /* A use. */
   }

SAME:;                         /* Label name space. */
}
```

398 A member of an enumeration is called an *enumeration constant*.

enumeration constant

Commentary

This defines the term *enumeration constant*.

C++

There is no such explicit definition in the C++ Standard (7.2p1 comes close), although the term *enumeration constant* is used.

Other Languages

The term *enumeration constant* is generic to most languages that contain enumerated types.

Coding Guidelines

It is important to use this terminology. Sometimes the terms *enumeration identifier* or *enumeration value* are used by developers. It is easy to confuse these terms with other kinds of identifiers, or values.

Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions. 399

Commentary

macro definition lasts until 1974
macro definition 1975
no significance after preprocessing directives deleted 132
translation phase 7 136

Macro names are different from all other names in that they exist in a single scope that has no nesting. Macro definitions cease to exist after translation phase 4, as do preprocessing directives. The so-called *semantic phase* is translation phase 7.

Common Implementations

A few translators tag tokens with the macros they were expanded from, if any, as an aid to providing more informative diagnostic messages.

Coding Guidelines

Some of these coding guidelines apply to the source code that is visible to the developer. In such cases macro names, if they appear in the visible source, need to be considered in their unexpanded form.

preprocessor directives syntax 1854
#undef 1976

The concept of scope implies a coding structure that does not really exist during preprocessing (use of conditional inclusion does not create a new scope). The preprocessor views its input as an unstructured (apart from preprocessing directives) sequence of preprocessing tokens, some of which have special meaning. While it is possible to use **`#define`**/**`#undef`** pairs to simulate a scope, this usage is not common in practice.

---

visible identifier scope

For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. 400

Commentary

This defines the terms *visible* and *scope*. Visibility here means visible using the mechanisms defined in the standard for looking up identifiers (that have been previously declared). The concept of scope, in C, is based on the textual context in which an identifier occurs in translation phase 7— so-called *lexical scoping* (sometimes called *static scoping*, as opposed to *dynamic scoping*, which is based on when an object is created during program execution). The scope of a named object is closely associated with its lifetime in many cases.

translation phase 7 136
lifetime of object 451

C++

3.3p1 *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope.*

3.3p4 *Local extern declarations (3.5) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions.*

If a name is in scope and is not hidden it is said to be visible. 3.3.7p5

Other Languages

The concept of scope exists in most programming languages. In Cobol, and some dialects of Basic, all variables have the same scope. Languages which make use of dynamic scoping include APL, Perl, Snobol 4, and Lisp (later versions of Lisp use static scoping; support for static scoping was introduced in version 5 of Perl). While being very flexible, dynamic scoping is not very runtime efficient because the actual object referenced has to be found during program execution; not being known at translation time, it is not possible to fix its address prior to program execution.

```
1  #include <stdlib.h>
2
3  int total = 0;
4
5  void f_1(void)
6  {
7  /*
8   * With dynamic scoping the current function call chain is important.
9   * If f_1 is called via f_2 then the local definition of total in f_2
10  * would be assigned to.  If f_1 was called directly from main, the
11  * file scope declaration of total would be assigned to.
12  */
13 total=1;
14 }
15
16 void f_2(void)
17 {
18 int total;
19
20 f_1();
21 }
22
23 int main(void)
24 {
25 if (rand() > 20)
26    f_1();
27 else
28    f_2();
29
30 return total;
31 }
```

In some languages the scope of an identifier may not be the same region of program text as the one it is visible in. For instance Pascal specifies that the scope of an identifier starts at the beginning of the block that contains its declaration, but it is only visible from the point at which it is declared.

```
1  int sum;
2
3  void pascal_scope_rules(void)
4  {
5  /*
6   * In Pascal the scope of the local declarations start here.  The scope of the
7   * nested declaration of sum starts here and hides the file scope declaration.
8   * But the nested declaration of sum does not become visible until after its'
9   * declaration.
10  */
11 int add_up = sum; /* In Pascal there is no identifier called sum visible here. */
```

```
12  int sum = 3;
13  }
```

Common Implementations

Implementations used to make use of the fact that identifiers were only visible (and therefore needed to be in their symbol table) within a scope to reduce their internal storage requirements; freeing up the storage used for an identifier's symbol table entry when the processing of the scope that contained its declaration was complete. Many modern implementations are not driven by tight storage restrictions and may keep the information for reasons of optimization or improved diagnostic messages.

Some implementations *exported* external declarations that occurred in block scope to file scope. For instance:

```
1   void f1(void)
2   {
3   /*
4    * Some translators make the following external linkage
5    * declaration visible at file scope...
6    */
7   extern int glob;
8   }
9
10  void f2(void)
11  {
12  /*
13   * ... which means that the following reference to glob refers
14   * to the one declared in function f1.
15   */
16  glob++;
17  }
18
19  void f3(void)
20  {
21  /*
22   * Linkage ensures that the following declaration always
23   * refers to the same glob object declared in function f1.
24   */
25  extern int glob;
26
27  glob++;
28  }
```

Coding Guidelines

Objects and functions are different from other entities in that it is possible to refer to them, via pointers, when the identifiers that designate them are not visible. While such anonymous accesses (objects referenced in this way are said to be *aliased*) can be very useful, but they can also increase the effort needed, by developers, to comprehend code. The issue of object aliases is discussed elsewhere.

alias analysis 1491
object 971 aliased

---

same identifier

Different entities designated by the same identifier either have different scopes, or are in different name spaces. 401

Commentary

The C Standard's meaning of same identifier is that the significant characters used to spell the identifier are identical. Differences in non-significant characters are not considered. Attempting to declare an identifier in the same scope and name space as another identifier with the same spelling is always a constraint violation.

C90

In all but one case, duplicate label names having the same identifier designate different entities in the same scope, or in the same name space, was a constraint violation in C90. Having the same identifier denote two different labels in the same function caused undefined behavior. The wording in C99 changed to make this case a constraint violation.

1725 label name unique

C++

The C++ Standard does not explicitly make this observation, although it does include wording (e.g., 3.6.1p3) that implies it is possible.

Coding Guidelines

The concept of name space is not widely appreciated by C developers. Given the guideline recommendation that identifier names be unique, independently of what name space they are in, there does not appear to be any reason for wanting to educate developers further on the concept of name space.

792.3 identifier reusing names

---

402 There are four kinds of scopes: function, file, block, and function prototype.

scope kinds of

Commentary

File scope is also commonly called *global scope*, not a term defined by the standard. Objects declared at file scope are sometimes called *global objects*, or simply *globals*.

Block scope is also commonly called *local scope*, not a term defined by the standard. Objects declared in block scope are sometimes called *local objects*, or simply *locals*. A block scope that is nested within another block scope is often called a *nested scope*.

The careful reader will have noticed a subtle difference between the previous two paragraphs. Objects were declared "at" file scope, while they were declared "in" block scope. Your author cannot find any good reason to break with this common developer usage and leaves it to English majors to rant against the irregular usage.

Identifiers defined as macro definitions are also said to have a scope.

1974 macro definition lasts until

C++

The C++ Standard does not list the possible scopes in a single sentence. There are subclauses of 3.3 that discuss the five kinds of C++ scope: function, namespace, local, function prototype, and class. A C declaration at file scope is said to have *namespace scope* or *global scope* in C++. A C declaration with block scope is said to have *local scope* in C++. Class scope is what appears inside the curly braces in a structure/union declaration (or other types of declaration in C++).

Given the following declaration, at file scope:

```
1  struct S {
2            int m; /* has file scope */
3                   // has class scope
4            } v;   /* has file scope */
5                   // has namespace scope
```

Other Languages

File and block scope are concepts that occur in many other languages. Some languages only allow definitions within functions to occur in the outermost block (e.g., Pascal); this may nor may not be equated to function scope. Function prototype scope is unique to C (and C++).

Coding Guidelines

Many developers are aware of file and block scope, but not the other two kinds of scope. Is anything to be gained by educating developers about these other scopes? Probably not. There are few situations where they are significant. These coding guidelines concentrate on the two most common cases— file and block scope (one issue involving function prototype scope is discussed elsewhere).

404 scope function
409 scope function prototype
409 prototype providing identifiers

(A *function prototype* is a declaration of a function that declares the types of its parameters.) 403

Commentary

This defines the term *function prototype*; it appears in parentheses because it is not part of the main subject of discussion in this subclause. A function prototype can occur in both a function declaration and a function definition. Function prototype scope only applies to the parameters in a function prototype that is purely a declaration. The parameters in a function prototype that is also a definition have block scope.

scope 409
function prototype
block scope 408
terminates

scope
function

A label name is the only kind of identifier that has *function scope*. 404

Commentary

labeled 1722
statements
syntax
goto 1787
statement

Labels are part of a mechanism (the **goto** statement) that can be used to change the flow of program execution. The C language permits arbitrary jumps to the start of any statement within a function (that contains the jump). To support this functionality label name visibility needs to cut across block boundaries, being visible anywhere within the function that defines them.

Other Languages

Few other languages define a special scope for labels. Most simply state that the labels are visible within the function that defines them, although a few (e.g., Algol 68) give labels block scope (this prevents jumps into nested blocks).

Languages that supported nested function definitions and require labels to be defined along with objects, often allow labels defined in outer function definitions to be referenced from functions defined within them.

Common Implementations

gcc supports taking the address of a label. It can be assigned to an object having a pointer type and passed as an argument in a function call. The label identifier still has function scope, but anonymous references to it may exist in other function scopes during program execution.

Coding Guidelines

This scope, which is specific to labels, is not generally known about by developers. Although they are aware that labels are visible throughout a function, developers tend not to equate this to the concept of a separate kind of scope. There does not appear to be benefit educating developers about its existence.

label
declared implicitly

It can be used (in a **goto** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a **:** and a statement). 405

Commentary

The C Standard provides no other mechanism for defining labels. Labels are the only entities that may, of necessity, be used before they are defined (a forward jump).

goto 1787
statement

Other Languages

Some languages allow labels to be passed as arguments in calls to functions and even assigned to objects (whose contents can then be **goto**ed.)

Pascal uses the **label** keyword to define a list of labels. Once defined, these labels may subsequently label a statement or be referenced in a **goto** statement. Pascal does not support declarations in nested blocks; so labels, along with all locally declared objects and types, effectively have function scope.

Common Implementations

gcc allows a label to appear as the operand of the **&&** unary operator (an extension that returns the address of the label).

identifier
scope determined
by declaration
placement

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). 406

Commentary

Every other identifier must also appear in a declarator before it is referenced. The C90 support for implicit declarations of functions, if none was currently visible, has been removed in C99. The textual placement of a declarator is the only mechanism available in C for controlling the visibility of the identifiers declared.

Other Languages

Languages that support a more formal approach to separate compilation have mechanisms for importing previously declared identifiers into a scope. Both Java and C++ support the **`private`** and **`public`** keywords, these can appear on a declarator to control the visibility of any identifiers it declares. Ada has a sophisticated separate compilation mechanism. Many other languages use very similar scoping mechanisms to those defined by the C Standard.

1810 translation unit syntax

Coding Guidelines

Some coding guideline documents recommend that the scope of identifiers be minimized. However, such a recommendation is based on a misplaced rationale. It is not an identifier's scope that should be minimized, but its visibility. For instance, an identifier at file scope may, or may not, be visible outside of the translation unit that defines it. Some related issues are discussed elsewhere.

1348 identifier definition close to usage

---

407 If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit.

file scope

Commentary

Structure and union members and any other identifiers declared within them, at file scope, also have file scope.

The return type on a function definition is outside of the outermost block associated with that definition. However, the parameters are treated as being inside it. Thus, any identifiers declared in the return type have file scope, while any declared in the parameter list have block scope.

408 block scope terminates

File scope may terminate at the end of the translation unit, but an identifier may have a linkage which causes it to be associated with declarations outside of that translation unit.

112 translation units communication between

Where the scope of an identifier begins is defined elsewhere.

416 tag scope begins
417 enumeration constant scope begins
418 identifier scope begins

C++

A name declared outside all named or unnamed namespaces (7.3), blocks (6.3), function declarations (8.3.5), function definitions (8.4) and classes (9) has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.1) and ends at the end of the translation unit that is its declarative region.

3.3.5p3

Other Languages

Nearly every other computer language supports some form of file scope declaration. A few languages, usually used in formal verification, do not allow objects to have file scope (allowing such usage can make it significantly more difficult to prove properties about a program).

In some languages individual identifiers do not exist independently at file scope— they must be part of a larger whole. For instance, in Fortran file scope objects are declared within **`common`** blocks (there can be multiple named common blocks, but only one unnamed common block). An entire common block (with all the identifiers it contains) needs to be declared within a source file, it is not possible to declare one single identifier (unless it is the only one declared by the common block). Other kinds of identifier groupings include *package* (Ada and Java), *namespace* (C++), *module* (Modula-2, Pascal), *cluster* (Clu), and *unit* (Borland Delphi). The underlying concept is the same, identifiers are exported and imported as a set.

Coding Guidelines

Some coding guidelines documents recommend that the number of objects declared at file scope be minimized. However, there have been no studies showing that alternative design/coding techniques would have a more worthwhile cost/benefit associated with their use.

The reasons for the declaration of an identifier to appear at file can depend on the kind of identifier being declared. Identifiers declared at file scope often appear in an order that depends on what they denote (e.g., macro, typedef, object, function, etc.). The issues associated with this usage are discussed elsewhere. The rest of this coding guideline subsection discusses the declaration of types, functions, and objects at file scope. Identifiers declared as types must appear at file scope if

external declaration 1810 syntax

- objects declared to have these types appear at file scope,
- objects declared to have these types appear within more than one function definition (having multiple, textual, declarations of the same type in different block scopes is likely to increase the cost of maintenance and C does not support the passing of types as parameters),
- other types, at files scope, reference these types in their own declaration.

Identifiers declared as functions must appear at file scope in those cases where two or more functions have a mutual calling relationship. This issue is discussed elsewhere.

function call 1026 recursive

Any program using file scope objects can be written in a form that does not use file scope objects. This can be achieved either by putting all statements in the function `main`, or by passing, what were file scope, objects as parameters. Are these solutions more cost effective than what they replaced? The following is a brief discussion of some of the issues.

The following are some of the advantages of defining objects at file scope (rather than passing the values they contain via parameters) include:

- Efficiency of execution. Accessing objects at file scope does not incur any parameter passing overheads. The execution-time efficiency and storage issues are likely to be a consideration in a freestanding environment, and unlikely to be an issue in a hosted environment.
- Minimizes the cost of adding new function definitions to existing source code. If the information needed by the new function is not available in a visible object, it will have to be passed as an argument. The function calling the new function now has a requirement to access this information, to pass it as a parameter. This requirement goes back through all call chains that go through the newly created function. If the objects that need to be accessed have file scope, there will be no need to add any new arguments to function calls and parameters to existing function definitions. In some cases using parameters, instead of file scope objects, can dramatically increase the number of arguments that need to be passed to many functions; it depends on how information flows through a program. If the flow is hierarchical (i.e., functions are called in a tree-like structure), it is straight-forward to pass information via parameters. If the data flow is not hierarchical, but like an undirected graph: with both `x` and `y` calling `a`, it is necessary for `p` to act as key holder for any information they need to share with `a`. The degree to which information flow and control flow follow each other will determine the ease, or complexity, of using parameters to access information. The further up the call chain the shared calling function, the greater the number of parameters that need to be passed through.

limit 288 parameters in definition

The following are some of the disadvantages of defining objects at file scope:

- Storage is used for the duration of program execution.
- There is a single instance of object. Most functions are not recursive, so separate objects for nested invocations of a function are not usually necessary.

function call 1026 recursive

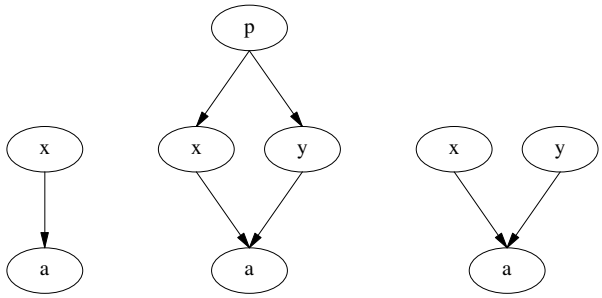


Figure 407.1: Some of the ways in which a function can be called— a single call from one other function; called from two or more functions, which in turn are all called by a single function; and called from two or more functions whose nearest shared calling function is not immediately above them.

- Reorganizing a program by moving function definitions into different translation units requires considering the objects they access. These may need to be given external linkage.
- Greater visibility of identifiers reduces the developer effort needed to access them, leading to a greater number of *temporary accesses* (the answer to the question, "how did all these unstructured accesses get into the source?" is, "one at a time").
- Information flow between functions is implicit. Developers need to make a greater investment in comprehending which calls cause which objects to be modified. Experience shows that developers tend to overestimate the reliability of their knowledge of which functions access which file scope identifiers.
- When reading the source of a translation unit, it is usually necessary to remember all of the file scope objects defined within it (locality of reference suggests that file scope identifiers referenced are more likely to be those defined in the current, rather than other, translation units). Reducing the number of file scope objects reduces the amount of information that needs to be in developers long-term memory.
- The only checks made on references to file scope objects is that they are visible and that the necessary type requirements are met. For issues, such as information flow, objects required to have certain values at certain points are not checked (such checking is in the realm of formal checking against specifications). Passing information via parameters does not guarantee that mistakes will not be made; but the need to provide arguments acts as a reminder of the information accessed by a function and the possible consequences of the call.

What is the algorithm for deciding whether to use parameters or file scope objects? The answer to this question involves many complex issues that are still poorly understood.

The argument that many programs exhibit faults because of the unconstrained use of objects at file scope, therefore use of parameters must be given preference, is too narrowly focused. Because it is the least costly solution, in terms of developer effort, file scope objects are more likely to be used than parameter passing. Given that many programs do not have long lifespans their maintenance costs are small, or non-existent. The many small savings accrued over incremental changes to many small programs over a relatively short lifespan may be greater than the total costs incurred from the few programs that are maintained over longer periods of time. If these savings are not greater than the costs, what is the size of the loss? Is it less than the value $saving - cost$ for the case in which developers have to invest additional effort in passing information via parameters? Without reliable evidence gathered from commercial development projects, these coding guidelines are silent on the topic of use file scope versus use of parameters.

Although they don't have file scope, the scope of macro names does terminate at the same point as identifiers that have file scope. Macro definitions are discussed elsewhere.

1974 macro definition lasts until
792 scope naming conventions

The issue of how an identifier's scope might interact with its spellings is discussed elsewhere.

Example

```
1  enum {E1, E2} /* File scope. */
2                f(enum {E3, E4} x) /* Block scope. */
3  { /* ... */ }
4
5  void g(c)
6  enum m {q, r} c; /* Block scope, in the list of parameter declarations. */
7  { /* ... */ }
```

---

block scope terminates

If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. 408

Commentary

This defines the term *block scope* and specifies where it terminates. Where the scope of an identifier begins is defined elsewhere.

tag 416 scope begins
enumeration 417 constant scope begins
identifier 418 scope begins

Structure and union members, and other identifiers declared within them (but not those defined as macro names), in block scope, also have block scope.

The reference to parameter declarations does not distinguish between those declared via a function prototype, or an old style function declaration. They both have block scope, denoted by the outermost pair of braces.

It is possible for an object or function declared in block scope to refer to the same object or function at file scope, or even in a different translation unit, through a process called *linkage*.

linkage 420

C++

3.3.2p1 *A name declared in a block (6.3) is local to that block. Its potential scope begins at its point of declaration (3.3.1) and ends at the end of its declarative region.*

3.3.2p2 *The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. . . . , else it ends at the outermost block of the function definition.*

Other Languages

A few languages, for instance Cobol, do not have any equivalent to block scope. In Java local variables declared in blocks follow rules similar to C. However, there are situations where the scope of a declaration can reach back to before its textual declaration.

Gosling[508] *The scope of a member declared in or inherited by a class type (8.2) or interface (9.2) is the entire declaration of the class or interface type.*

```
class Test {
   test() { k = 2; }
   int k;
   }
```

Common Implementations

Parameters in a function definition having block scope, rather than function prototype scope causes headache for translator implementors. The traditional single pass, on-the-fly translation process does not know that the parameters have block scope until an opening curly bracket, a semicolon, or a type (for old style function definitions) is seen during syntax analysis.

10 implementation single pass

Coding Guidelines

When should identifiers be defined with block scope?

Possible benefits of declaring functions in block scope is to remove the overhead associated with having to **#include** the appropriate header (in translation environments having limited storage capacity this can be a worthwhile saving), and to allow the source of a function definition to be easily copied to other source files (the cut-and-paste model of software development). Possible benefits of defining objects in block scope is that storage only needs to be allocated when the block containing it is executed, and objects within each block can be thought about independently (i.e., developers do not need to remember information about objects defined in other blocks).

Experience shows that when writing programs, developers often start out giving most objects block scope. It is only when information needs to be shared between different functions that the possible use of file scope, as opposed to parameters, needs to be considered. This issue is discussed in the previous C sentence.

407 file scope

It is rare for a declaration of an identifier denoting a typedef, tag, or enumeration constant to occur in block scope. Data structures important enough to warrant such a declaration are normally referenced by more than one function. Such usage could be an indicator that a function is overly long and needs to be broken up into smaller functions.

286 identifiers number in block scope

The same coding guideline issues might also apply to macro definitions. That is, macro names that are only used within a single function definition be defined at the start of that function (along with other definitions local to that function). However, macros do not have block scope and existing practice is for macro definitions to be located at the start of source files (and this is where developers learned to expect to find them). Calculating whether there is a worthwhile cost/benefit in using **#define**/**#undef** pairs to emulate a block scope is likely to be difficult and these coding guidelines are silent on the issue.

macro definition emulate block scope
1854 preprocessor directives syntax

Example

```
1  void DR_035(c)
2  enum m{q, r} /* m, q and r all have block scope, not file scope. */
3               c;
4  { /* ... */ }
5
6  void f(void)
7  {
8  /*
9   * The following function declaration declares a type in its
10  * return type.  This has block scope, effectively rendering
11  * calls to this function as undefined behavior.
12  */
13 extern struct {int i;} undefined_func(void);
14 }
```

---

409 If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.

scope function prototype

Commentary

This type of scope was created by the C Committee for the C90 Standard (it was not in the original base document). This kind of scope is needed because the Committee decide to permit declarations of the form

1 base document

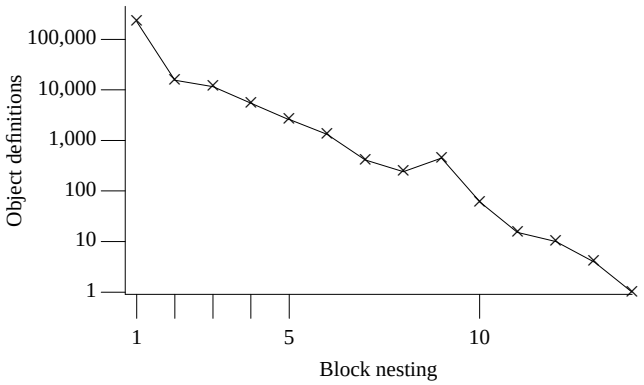


Figure 408.1: Number of object declarations appearing at various block nesting levels (level 1 is the outermost block). Based on the translated form of this book's benchmark programs.

prototype 409 providing identifiers

`extern void f(int x);`, where an identifier is specified for the parameter. Some developers believe that use of parameter names provides a benefit (this issue is discussed elsewhere).

A function declaration that defines a structure or union tag, or an anonymous structure or union type, in its function prototype scope renders the identifier it declares uncallable within the current translation unit (unless an explicit cast is applied to the function declarator at the point of call). This is because any calls to it cannot contain arguments whose types are compatible with the types of the parameters (which are declared using types that are only visible inside the function prototype scope). In the case of enumeration types declared in a function prototype scope, the identifier declared is callable (any arguments having an integer type will be converted to the enumerated types compatible integer type).

Other Languages

C (and C++) is unique in having this kind of scope. Other languages tend to simply state that the identifiers within the matching parentheses must be unique and do not allow new types to be defined between them, removing the need to create a scope to make use of the associated functionality to correctly process the parameter declarations.

Coding Guidelines

Tags, enumeration constants and parameter identifiers declared in function prototype scope are not visible outside of that scope and there are no mechanisms available for making them visible. Such usage serves no useful purpose since function prototype scope hides them completely. Anonymous structure and union types declared in function prototype scope do nothing but render the declared function unreferenceable.

prototype providing identifiers

The identifiers appearing in a prototype essentially fill the same role as comments, i.e., they may provide a reader of the source with useful information. The only reason that identifiers appear in such declarations may be because the author cut-and-pasted from the function definition. The presence of identifiers slightly increases the probability that a match against a defined macro will occur, but the consequences are either harmless or a syntax error. There does not appear to be a worthwhile cost/benefit in a guideline recommendation for or against this usage.

Example

```
1  /*
2   * Declaring a new type in a function prototype scope renders that
3   * function uncallable.  No other function can be compatible with it.
4   */
5  extern void uncallable(struct s_tag{int m1;} p);
6
7  extern void call_if_compatible_int(enum {E1, E2} p);
```

410 If an identifier designates two different entities in the same name space, the scopes might overlap.

scope overlapping

Commentary

Or they will be disjoint, or the translation unit contains a constraint violation.

Note that if a tag denoting an incomplete type is visible, then another declaration of that identifier in a different scope in the tag name space introduces a new type; it does not complete the previously visible incomplete type.

1459 tag declarations different scope

C90

This sentence does not appear in the C90 Standard, but the situation it describes could have occurred in C90.

C++

The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. 3.3p1

Other Languages

Languages that support at least two forms of scope (file and block) invariably allow identifiers declared in them to have overlapping scopes.

Coding Guidelines

Why would a developer want to use the same name to designate different entities in the same name space with overlapping scopes? Such usage can occur accidentally; for instance, when a block scope identifier has the same name as a file scope identifier that is beyond the developer's control (e.g., in a system header). Having the same identifier designate different entities, irrespective of name space and scope, is likely to be a potential cause of confusion to developers.

When two identifiers in the same name space have overlapping scopes, it is possible for a small change to the source to result in a completely unexpected change in behavior. For instance, if the identifier definition in the inner scope is deleted, all references that previously referred to that, inner scoped, identifier will now refer to the identifier in the outer scope. Deleting the definition of an identifier is usually intended to be accompanied by deletions of all references to it; the omission of a deletion usually generates a diagnostic when the source is translated (because of a type mismatch). However, if there is an alternative definition, with a compatible type, for an access to refer to, translators are unlikely to issue a diagnostic.

Dev 792.3 Identifiers denoting objects, functions, or types need only be compared against other identifiers having, or returning, a scalar type.

Example

```
1  extern int total_valu;
2  extern struct T {
3                   int m1;
4                  } data_fields;
5
6  void f(void)
7  {
8  float total_valu = 0.0;
9  int data_fields = 0;
10 }
11
```

```
12  void g(void)
13  {
14  int X;
15
16     {
17     int Y = X; /* Scope of second X not yet opened. */
18     int X;
19     }
20  }
```

Once an outer identifier has been hidden by the declaration in an inner scope, it is not possible to refer, by name, to the outer object.

---

scope inner
scope outer

If so, the scope of one entity (the *inner scope*) will be a strict subset of the scope of the other entity (the *outer scope*). 411

Commentary

This defines the terms *inner scope* and *outer scope*. The term *nested scope* is also commonly used to describe a scope that exists within another scope (often a block scope nested inside another block scope).

file scope accessing hidden names

This C statement only describes the behavior of identifiers. It is possible to refer to objects in disjoint scopes by using pointers to them. It is also possible, through use of linkage, for an identifier denoting an object at file scope to be visible within a nested block scope even though there is another declaration of the same identifier in an intervening scope.

```
1   extern int glob;
2
3   void f1(void)
4   {
5   int glob = 0;
6
7      {
8      extern int glob; /* Refers to same object as file scope glob. */
9
10     glob++;
11     }
12  /* Visible glob still has value 0 here. */
13  }
```

C++

The C observation can be inferred from the C++ wording.

3.3p1 *In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.*

Other Languages

This terminology is common to most programming languages.

Within the inner scope, the identifier designates the entity declared in the inner scope; 412

Commentary

identifier 418 scope begins

The scopes of identifiers, in C, do not begin at the opening curly brace of a compound block (for block scope). It is possible for an identifier to denote different entities within the same compound block.

```
1   typedef int I;
2
3   void f(void)
```

```
4   {
5   I I; /* The inner scope begins at the second I. */
6   }
```

Coding Guidelines

Developers are unlikely to spend much time thinking about where the scope of an identifier starts. A cost effective simplification is to think of an identifier's scope being the complete block that contains its declaration. One of the rationales for the guideline recommendation dealing with reusing identifier names is to allow developers to continue to use this simplification.

792.3 identifier reusing names

---

413 the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

outer scope identifier hidden

Commentary

This defines the term *hidden*. The entity is hidden in the sense that the identifier is not visible. If the entity is an object, its storage location will still be accessible through any previous assignments of its address to an object of the appropriate pointer type. If such identifiers are file scope objects, they may also be accessible via, developer-written, functions that access them (possibly returning, or modifying, their value).

There is only one mechanism for directly accessing the outer identifier via its name, and that only applies to objects having file scope.

411 file scope accessing hidden names

C++

The C rules are a subset of those for C++ (3.3p1), which include other constructs. For instance, the scope resolution operator, `::`, allows a file scope identifier to be accessed, but it does not introduce that identifier into the current scope.

Other Languages

The hiding of identifiers by inner, nested declarations is common to all block-structured languages. Some languages, such as Lisp, base the terms *inner scope* and *outer scope* on a time-of-creation basis rather than lexically textual occurrence in the source code.

Coding Guidelines

The discussion of the impact of scope on identifier spellings is applicable here.

792 scope naming conventions

Example

It is still possible to access an outer scope identifier, denoting an object, via a pointer to it:

```
1   void f(void)
2   {
3   int i,
4       *pi = &i;
5
6       {
7       int i = 3;
8
9       i += *pi; /* Assign the value of the outer object i, to the inner object i. */
10      }
11  }
```

Usage

In the translated form of this book's benchmark programs there were 1,945 identifier definitions (out of 270,394 identifiers defined in block scope) where an identifier declared in an inner scope hid an identifier declared in an outer block scope.

---

414 Unless explicitly stated otherwise, where this International Standard uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

Commentary

The C Standard is a definition of a computer language in a stylized form of English. It is not intended as a tutorial. As such, it is brief and to the point. This wording ensures that uses of the term *identifier* are not misconstrued. For instance, later wording in the standard should not treat the declaration of `glob` as being a prior declaration in the context of what is being discussed.

prior declaration not 429

```
1  struct glob {
2              int mem;
3              };
4
5  extern int glob;
```

C90

There is no such statement in the C90 Standard.

C++

There is no such statement in the C++ Standard (which does contain uses of *identifier* that refer to its syntactic form).

Coding Guidelines

Coding guidelines are likely to be read by inexperienced developers. Particular guidelines may be read in isolation, relative to other guidelines. Being brief is not always a positive attribute for them to have. A few additional words clarifying the status of the identifier referred to can help confirm the intent and reduce possible, unintended ambiguities.

---

scope same

Two identifiers have the *same scope* if and only if their scopes terminate at the same point. 415

Commentary

incomplete type completed by 550

This defines the term *same scope* (it is used in the description of incomplete types). The scope of every identifier, except labels, starts at a different point than the scope of any other identifier in the source file. Scopes end at well-defined boundaries; the closing **}** of a block, the closing **)** of a prototype, or on reaching the end of the source file, the end of an *iteration-statement*, loop body, the end of a *selection-statement*, or the end of a substatement associated with a selection statement.

block iteration statement 1768
block loop body 1769
block selection statement 1741
block selection sub-statement 1742

C90

Although the wording of this sentence is the same in C90 and C99, there are more blocks available to have their scopes terminated in C99. The issues caused by this difference are discussed in the relevant sentences for iteration-statement, loop body, a *selection-statement* a substatement associated with a selection statement.

block iteration statement 1768
block loop body 1769
block selection statement 1741
block selection sub-statement 1742

C++

The C++ Standard uses the term *same scope* (in the sense "in the same scope", but does not provide a definition for it. Possible interpretations include using the common English usage of the word *same* or interpreting the following wording

3.3p1 *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope. To determine the scope of a declaration, it is sometimes convenient to refer to the potential scope of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.*

to imply that the scope of the first declaration of `a` is not the same as the scope of `b` in the following:

```
{
int a;
int b;
   {
   int a;
   }
}
```

Other Languages

A few languages define the concept of same scope, often based on where identifiers are declared not where their scope ends.

Example

In the following example the identifiers `X` and `Y` have the same scope; the identifier `Z` is in a different scope.

```
void f(void)
{
int X;
int Y;

   {
   int X; /* A different object named X. */
   int Z;

   } /* Scope of X and Z ended at the } */
}    /* Scope of X and Y ended at the } */

void g(void)
{
for (int index = 0; index < 10; index++)
   {
   int index;
   /* Scope of second index ends here. */ }
/* Scope of first one ends here.    */ }
```

---

416 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag.

tag scope begins

Commentary

The scope of some identifiers does not start until their declarator is complete. Applying such a rule to tags would prevent self-referencing structure and union types (i.e., from having members that pointed at the type currently being defined).

418 identifier scope begins

```
struct S1_TAG {
               struct S1_TAG *next;
               };
```

However, just because a tag is in scope and visible does not mean it can be referenced in all of the contexts that a tag can appear in. A tag name may be visible, but denoting what is known as an *incomplete type*.

550 incomplete type completed by

```
struct S2_TAG {
               char mem1[sizeof(struct S2_TAG)]; /* Constraint violation. */
               };

/*
 * Here E_TAG becomes visible once the opening { is reached.
 * However, the decision on the size of the integer type chosen may
```

```
 * require information on all of the enumeration constant values and
 * its type is incomplete until the closing }.
 */
enum E_TAG {E1 = sizeof(enum E_TAG)};
```

C++

The C++ Standard defines the term *point of declaration* (3.3.1p1). The C++ point of declaration of the identifier that C refers to as a tag is the same (3.3.1p5). The scope of this identifier starts at the same place in C and C++ (3.3.2p1, 3.3.5p3).

Coding Guidelines

incom-plete type completed by 550

The scope of a tag is important because of how it relates to completing a previous incomplete structure or union type declaration. This issue is discussed in more detail elsewhere.

---

enumera-tion constant scope begins

Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. 417

Commentary

enumeration specifier syntax 1439

The definition of an enumerator constant can include a constant expression that specifies its value. The scope of the enumeration constant begins when the following comma or closing brace is encountered. This enables subsequent enumerators to refer to ones previously defined in the same enumerated type definition.

C++

3.3.1p3

The point of declaration for an enumerator is immediately after its `enumerator-definition`*. [Example:*

```
const int x = 12;

{ enum { x = x }; }
```

Here, the enumerator `x` *is initialized with the value of the constant* `x`*, namely 12.]*

In C, the first declaration of x is not a constant expression. Replacing it by a definition of an enumeration of the same name would have an equivalent, conforming effect in C.

Coding Guidelines

identifier reusing names 792.3

The example below illustrates an extreme case of the confusion that can result from reusing identifier names. The guideline recommendation dealing with reusing identifier names is applicable here.

Example

```
enum {E = 99};

void f(void)
{
enum { E = E + 1  /* Original E still in scope here. */
                , /* After comma, new E in scope here, with value 100. */
       F = E};    /* Value 100 assigned to F. */
}
```

---

identifier scope begins

Any other identifier has scope that begins just after the completion of its declarator. 418

Commentary

The other identifiers are objects, typedefs, and functions. For objects' definitions, the declarator does not include any initializer that may be present. A declarator may begin the scope of an identifier, but subsequent declarators in the same scope for the same identifier may also appear (and sometimes be necessary) in some cases.

1547 declarator syntax

550 incomplete type completed by

C++

The C++ Standard defines the potential scope of an identifier having either local (3.3.2p1) or global (3.3.5p3) scope to begin at its *point of declaration* (3.3.1p1). However, there is no such specification for identifiers having function prototype scope (which means that in the following declaration the second occurrence of `p1` might not be considered to be in scope).

```
1  void f(int p1, int p2[sizeof(p1)]);
```

No difference is flagged here because it is not thought likely that C++ implementation will behave different from C implementations in this case.

Other Languages

Some languages define the scope of an identifier to start at the beginning of the block containing its declaration. This design choice can make it difficult for a translator to operate in a single pass. An identifier from an outer scope may be referenced in the declaration of an object; a subsequent declaration in the same block of an identifier with the same name as the referenced outer scope one invalidates the reference to the previous object declaration and requiring the language translator to change the associated reference.

In some cases the scope of an identifier in Java can extend to before its point of declaration.

408 block scope terminates

Coding Guidelines

Cases such as the one given in the next example, where two entities share the same name and are visible in different portions of the same block are covered by the guideline recommendation dealing with reusing identifier names.

792.3 identifier reusing names

Example

```
1  void f(void)
2  {
3  enum {c, b, a};
4  typedef int I;
5
6     {
7     I I; /*
8           * The identifier object does not become visible until the ;
9           * is reached, by which time the typedefed I has done its job.
10          */
11
12    int a[a]; /*
13                * Declarator completed after the closing ], number of
14                * elements refers to the enumeration constant a.
15                */
16
17    struct T {struct T *m;} x = /* declarator complete here. */
18                                {&x};
19    }
20 }
```

419 **Forward references:** declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), name spaces of identifiers (6.2.3), macro replacement (6.10.3), source file inclusion (6.10.2), statements (6.8).

6.2.2 Linkages of identifiers

linkage

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[21)] 420

Commentary

This defines the term *linkage*. It was introduced into C90 by the Committee as a means of specifying a model of separate compilation for C programs.

It is intended that the different scopes mentioned apply to file scopes in both the same and different translation units (block scopes can also be involved in the case of tags). The need to handle multiple declarations, in the same and different translation units, of the same file scope objects was made necessary by the large body of existing source code that contained such declarations. Linkage is the mechanism that makes it possible to translate different parts of a program at different times, each referring to objects and functions defined elsewhere.

program 107
not translated at same time

Rationale

The definition model to be used for objects with external linkage was a major C89 standardization issue. The basic problem was to decide which declarations of an object define storage for the object, and which merely reference an existing object. A related problem was whether multiple definitions of storage are allowed, or only one is acceptable. Pre-C89 implementations exhibit at least four different models, listed here in order of increasing restrictiveness:

Common Every object declaration with external linkage, regardless of whether the keyword extern appears in the declaration, creates a definition of storage. When all of the modules are combined together, each definition with the same name is located at the same address in memory. (The name is derived from common storage in Fortran.) This model was the intent of the original designer of C, Dennis Ritchie.

Relaxed Ref/Def The appearance of the keyword extern in a declaration, regardless of whether it is used inside or outside of the scope of a function, indicates a pure reference (ref), which does not define storage. Somewhere in all of the translation units, at least one definition (def) of the object must exist. An external definition is indicated by an object declaration in file scope containing no storage class indication. A reference without a corresponding definition is an error. Some implementations also will not generate a reference for items which are declared with the extern keyword but are never used in the code. The UNIX operating system C compiler and linker implement this model, which is recognized as a common extension to the C language (see §K.5.11). UNIX C programs which take advantage of this model are standard conforming in their environment, but are not maximally portable (not strictly conforming).

Strict Ref/Def This is the same as the relaxed ref/def model, save that only one definition is allowed. Again, some implementations may decide not to put out references to items that are not used. This is the model specified in K&R .

Initialization This model requires an explicit initialization to define storage. All other declarations are references.

Table 420.1: Comparison of identifier linkage models

Model	File 1	File 2
common	`extern int I; int main() { I = 1; second(); }`	`extern int I; void second() { third( I ); }`
Relaxed Ref/Def	`int I; int main() { I = 1; second(); }`	`int I; void second() { third( I ); }`
Strict Ref/Def	`int I; int main() { I = 1; second(); }`	`extern int I; void second() { third( I ); }`
Initializer	`int I = 0; int main() { I = 1; second(); }`	`int I; void second() { third( I ); }`

C++

The C++ Standard also defines the term *linkage*. However, it is much less relaxed about multiple declarations of the same identifier (3.3p4).

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

Other Languages

Few languages permit multiple declarations of the same identifier in the same scope. The mechanisms used by different languages to cause an identifier to refer to the same object, or function, in separately translated source files varies enormously. Some languages, such as Ada and Java, have a fully defined separate compilation mechanism. Fortran requires that shared objects be defined in an area known as a common block. (The semantics of this model was used by some early C translators.) Other languages do not specify a separate compilation mechanism and leave it up to the implementation to provide one.

1810 translation unit syntax

Fortran specifies the common block construct as the mechanism by which storage may be shared across translation units. Common blocks having the same name share the same storage. The declarations within a common block are effectively offsets into that storage. There is no requirement that identifiers within each common block have the same name as identifiers at the corresponding storage locations of matching common blocks.

```
1          SUBROUTINE A
2          COMMON /XXX/ IFRED
3          END
4
5          SUBROUTINE B
6          COMMON /XXX/ JIM
7    C     IFRED and JIM share the same storage
8          END
```

Common Implementations

Early implementations provided a variety of models for deducing which declarations referred to the same object, as described in the previous Rationale discussion.

Coding Guidelines

The term *linkage* is not generally used by developers. Terms such as *externally visible* (or just *external*) and *not externally visible* (or *not external*, *only visible within one translation unit*) are used by developers when discussing issues covered by the C term *linkage*. Is it worthwhile educating developers about linkage and how it applies to different entities? From the developers point of view, the most important property is whether an object can be referenced from more than one translation unit. The common usage terms *external* and *not external* effectively describe the two states that are of interest to developers. Unless a developer wants to become an expert on the C Standard, the cost/benefit of learning how to apply the technically correct terminology (*linkage*) is not worthwhile.

linkage educating developers

In many ways following the guideline recommendation dealing with having a single point of declaration for each identifier often removes the need for developers to think about linkage issues.

422.1 identifier declared in one file

---

421 There are three kinds of linkage: external, internal, and none.

linkage kinds of

Commentary

The linkage *none* is sometimes referred to (in this standard and by developers) as *no linkage*.

C++

The C++ Standard defines the three kinds of linkage: external, internal, and no linkage. However, it also defines the concept of *language linkage*:

All function types, function names, and variable names have a language linkage. [Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here.

7.5p1

For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc.] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

Other Languages

The term *linkage* is unique to C (and C++). The idea behind it— of making identifiers declared in one separately translated source file visible to other translated files— is defined in several different ways by other languages.

Coding Guidelines

The term *external* is well-known to developers; some are also aware of the term *internal*. However, the fact that they relate to a concept called *linkage*, and that there is a third *none*, is almost unknown. The term *visible* is also sometimes used with these terms (e.g., *externally visible*).

---

object external linkage denotes same function external linkage denotes same

In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. 422

Commentary

same object 640 have compatible types

translation phase 139 8

The phrase *particular identifier* means identifiers spelled with the same character sequence (i.e., ignoring any nonsignificant characters). Other wording in the standard requires that the types of the identifiers be compatible (otherwise the behavior is undefined). The process of making sure that particular identifiers with external linkage refer to the same object, or function, occurs in translation phase 8.

C++

The situation in C++ is complicated by its explicit support for linkage to identifiers whose definition occurs in other languages and its support for overloaded functions (which is based on a function's signature (1.3.10) rather than its name). As the following references show, the C++ Standard does not appear to explicitly specify the same requirements as C.

3.2p3 *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is used.*

C++ 1350 one definition rule

Some of the consequences of the C++ *one definition rule* are discussed elsewhere.

3.5p2

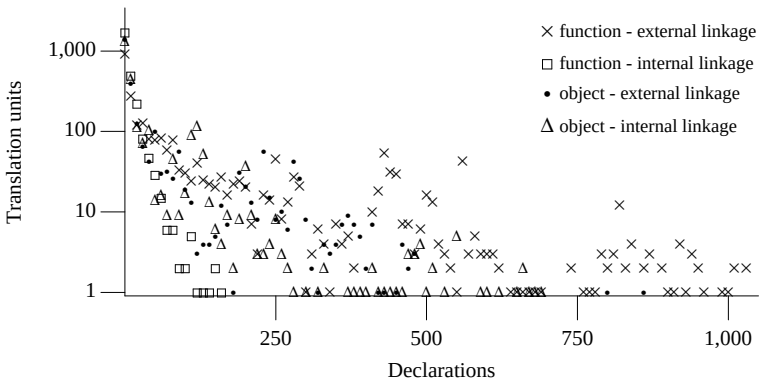


Figure 421.1: Number of translation units containing a given number of objects and functions declared with internal and external linkage (excluding declarations in system headers). Based on the translated form of this book's benchmark programs.

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

— When a name has external linkage, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.

At most one function with a particular name can have C language linkage. 7.5p6

Common Implementations

As a bare minimum, translators have to write information on the spelling of identifiers having external linkage to the object code file produced from each translation unit. Some translators contain options to write out information on other identifiers. This might be used, for instance, for symbolic debugging information.

Most linkers (the tool invariably used) simply match up identifiers (having external linkage) in different translated translation units with the same spelling (the spelling that was written to the object file by the previous phase, which will have removed nonsignificant characters). While type information is sometimes available (e.g., for symbolic debugging), most linkers ignore it. Very few do any cross translation unit type checking. The commonly seen behavior is to use the start address of the storage allocated (irrespective of whether an object or function is involved). 140 linkers

Issues, such as identifiers not having a definition in at least one of the translation units, or there being more than one definition of the same identifier, are discussed elsewhere. 1818 external linkage
exactly one external definition

Coding Guidelines

There is no requirement for a translator to generate a diagnostic message when a particular identifier is declared using incompatible types in different translation units. The standard defines no behavior for such usage and accessing such identifiers results in undefined behavior. A translator is not required to, and most don't, issue any diagnostic messages as a result of requirements in translation phase 8. 640 same object
have compatible types
139 translation phase
8

Because of the likely lack of tool support for detecting instances of incompatible declarations across translations units, these guidelines recommend the use of code-structuring techniques that make it impossible to declare the same identifier differently in different translation units. The lack of checking by translators, and other tools, then becomes a nonissue. The basic idea is to ensure that there is only ever one textual instance of the declaration of an identifier. This can never be different from itself, assuming any referenced macro names or typedefs are also the same. The issue then becomes how to organize the source to use this single textual declaration.

The simplest way of organizing source code to use single declarations is to place these declarations in header files. These header files can be included wherever their contents need to be referenced. A consequence of this guideline is that there are no visible external declarations in `.c` source files.

Cg 422.1

Only one of the source files translated to create a program image shall contain the textual declaration of a given identifier having file scope.

The C Standard requires all identifiers to be explicitly declared before they are referenced. However, this requirement is new in C99. The following guideline recommendation is intended to cover those cases where the translator used does not support the current standard, or where the translator is running in some C90 compatibility mode. 1000 operator ()

Cg 422.2

A source file that references an identifier with external linkage shall **`#include`** the header file that contains its textual declaration.

header name same as .c file
header name 918
external linkage 1818
exactly one external definition

A commonly used convention is for the `.h` file, containing the external declarations of identifiers defined in some `.c` file, to have the same header name as the `.c` file. The issue of ensuring that there is a unique definition for every referenced identifier is discussed elsewhere.

Example

The types of the object `array` in the following two files are incompatible.

file_1.c

```
1 extern int glob;
2 extern char array[10]; /* An array of 10 objects of type char. */
```

file_2.c

```
1 extern float glob;
2 extern char *array; /* A pointer to one or more objects of type char. */
```

Usage

A study of 29 Open Source programs by Srivastava, Hicks, Foster and Jenkins[1277] found 1,161 identifiers with external linkage, referenced in more than one translation unit, that were not declared in a header, and 809 instances where a header containing the declaration of a referenced identifier was not **#include**d (i.e., the source file contained a textual external declaration of the identifier).

---

identifier same if internal linkage

Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. 423

Commentary

tentative definition 1849
same object 640
have compatible types

This describes the behavior for the situation, described elsewhere, where there is more than one declaration of the same identifier with internal linkage in a translation unit. Other wording in the standard requires that the types of the identifiers be compatible

An object, defined with internal linkage in a translation unit, is distinct from any other object in other translation units, even if the definition of an identifier, denoting an object, in a different translation unit has exactly the same spelling (the linkage of the object in the other translation units is irrelevant).

Coding Guidelines

Why would a developer want to have more than one declaration of an identifier, with internal linkage, in one translation unit? There is one case where multiple declarations of the same object, with internal linkage, are necessary— when two or more object definitions recursively refer to each other (such usage is rare). There is also one case where multiple declarations of the same function, with internal linkage, are required—when two or more functions are in a recursive call chain. In this case some of the functions must be declared before they are defined. While it may be necessary to create type definitions that are mutually recursive, the declared identifiers have *none* linkage.

function call 1026
recursive
EXAMPLE 1474
mutually referential structures

Some coding guideline documents recommend that all functions defined in a translation unit be declared at the beginning of the source file, along with all the other declarations of file scope identifiers.

The rationale given for a guideline recommendation that identifiers with external linkage have a single textual declaration included a worthwhile reduction in maintenance costs. This rationale does not apply to declarations of identifiers having internal linkage, because translators are required to diagnose any type incompatibilities in any duplicate declarations and duplicate declarations are not common.

identifier 422.1
declared in one file

Example

```
1 struct T {struct T *next;};
2
3 static struct T q;
4
```

```
5   static struct T p = { &q }; /* Recursive reference. */
6   static struct T q = { &p }; /* Recursive reference. */
7
8   static void g(int);
9
10  static void f(int valu)
11  {
12  if (valu--)
13     g(valu);
14  }
15
16  static void g(int valu)
17  {
18  if (--valu)
19     f(valu);
20  }
```

424 Each declaration of an identifier with *no linkage* denotes a unique entity.

no linkage identifier declaration is unique

Commentary

Here the phrase *no linkage* is used to mean that the linkage is *none*. The term *no linkage* is commonly used to have this meaning. The only entities that can have a linkage other than linkage *none* are objects and functions, so identifiers denoting all other entities have *no linkage*.

Tags are the only identifiers having no linkage that may be declared more than once in the same scope. Other wording in the standard makes it a constraint violation to have more than one such identifier with the same name in the same scope and name space.

432 identifier no linkage
475 incomplete types
1350 declaration only one if no linkage

C++

The C++ *one definition rule* covers most cases:

> *No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.* 3.2p1

However, there is an exception:

> *In a given scope, a* **typedef** *specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [Example:* 7.1.3p2
>
> ```
> typedef struct s { /* ... */ } s;
> typedef int I;
> typedef int I;
> typedef I I;
> ```
>
> *—end example]*

Source developed using a C++ translator may contain duplicate typedef names that will generate a constraint violation if processed by a C translator.

The following does not prohibit names from the same scope denoting the same entity:

> *A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:* 3.5p2
>
> *— When a name has no linkage, the entity it denotes cannot be referred to by names from other scopes.*

This issue is also discussed elsewhere.

1350 declaration only one if no linkage

static internal linkage

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **static**, the identifier has internal linkage.[22)] 425

Commentary

no linkage 435 block scope object storage-class 1364 specifier syntax declarator 1547 syntax

If the same declaration occurs in block scope the identifier has no linkage. The keyword **static** is overworked in the C language. It is used to indicate a variety of different properties.

C++

3.5p3

A name having namespace scope (3.3.5) has internal linkage if it is the name of

— an object, reference, function or function template that is explicitly declared ***static*** *or,*

— an object or reference that is explicitly declared ***const*** *and neither explicitly declared* ***extern*** *nor previously declared to have external linkage; or*

```
1  const int glob; /* external linkage */
2                  // internal linkage
```

identifier 422.1 declared in one file identifier 1818.1 definition shall #include

Adhering to the guideline recommendations dealing with textually locating declarations in a header file and including these headers, ensures that this difference in behavior does not occur (or will at least cause a diagnostic to be generated if they do).

Other Languages

Some languages specify that all identifiers declared at file scope are not externally visible unless explicitly stated otherwise (i.e., the identifiers have to be explicitly exported through the use of some language construct).

Coding Guidelines

linkage 420 educating developers

Developers often think in terms of the keyword **static** limiting the visibility of an identifier to a single translation unit. This describes the effective behavior, but not the chain of reasoning behind it.

extern identifier linkage same as prior declaration

For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible,[23)] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. 426

Commentary

The C committee were faced with a problem. Historically, use of the keyword **extern** has been sloppy. Different declaration, organizational styles have been used to handle C's relaxed, separate compilation model. This sometimes resulted in multiple declarations of the same identifier using the storage-class specifier **extern**. Allowing the linkage of an object declared using the storage-class specifier **extern** to match the linkage of any previous declaration maintained the conformance status of existing (when C90 was written) source code.

While there may be more than one declaration of the same identifier in a translation unit, if it is used in an expression a definition of it is required to exist. If the identifier has internal linkage, a single definition of it must exist within the translation unit, and if it has external linkage there has to be exactly one definition of it somewhere in the entire program.

definition 1812 one external external 1818 linkage exactly one external definition

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

Coding Guidelines

While an identifier might only be textually declared in a single header file, that header may be **#include**d more than once when a source file is translated. A consequence of this multiple inclusion is that the same identifier can be declared more than once during translation (because it is a tentative definition). 1849 tentative definition

An identifier declared using the storage-class specifier **static** that is followed, in the same translation unit, by another declaration of the same identifier containing the storage-class specifier **extern** is considered to be harmless.

Example

```
1  static int si;
2  extern int si; /* si has internal linkage. */
3
4  static int local_func(void)
5  { /* ... */ }
6
7  void f(void)
8  {
9  extern int local_func(void);
10 }
11
12 static int glob; /* Declaration 1 */
13
14 void DR_011(void)
15 {
16 extern int glob; /* Declaration 2.  At this point Declaration 1 is visible. */
17
18    {
19    /*
20     * The following declaration results in an identifier that refers
21     * to the same object having both internal and external linkage.
22     */
23    extern int glob; /* Declaration 3.  Declaration 2 is visible here. */
24    }
25 }
```

---

427 21) There is no linkage between different identifiers. footnote 21

Commentary

Nor is there any linkage between identifiers in different name spaces; only identifiers in the *normal* name space can have linkage. Identifiers whose spelling differ in significant characters are different identifiers. The number of significant characters may depend on the linkage of the identifier. 282 internal identifier significant characters 283 external identifier significant characters

C++

The C++ Standard says this the other way around.

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope: 3.5p2

Coding Guidelines

Translators that support a limited number of significant characters in identifiers may create a linkage where none was intended. For instance, two different identifiers (when compared using all of the characters appearing in their spelling) with external linkage in different translation units may end up referring to each other because the translator used does not compare characters it considers nonsignificant. This issue is discussed elsewhere. 792 identifier number of characters

Example

Because there is no linkage between different identifiers, the following program never outputs any characters:

```
#include <stdio.h>

const int i = 0;
const int j = 0;

int main(void)
{
if (&i == &j)
   printf("Linkage exists between different identifiers\n");
return 0;
}
```

---

footnote 22

22) A function declaration can contain the storage-class specifier **static** only if it is at file scope; see 6.7.1. 428

Commentary

This sentence appears in a footnote and as such has no normative meaning. However, there is other wording in the standard that renders use of the static storage-class at block scope to be undefined behavior.

block scope 1371
storage-class use

C++

7.1.1p4 *There can be no* ***static*** *function declarations within a block, . . .*

This wording does not require a diagnostic, but the exact status of a program containing such a usage is not clear.

A function can be declared as a static member of a **class** (**struct**). Such usage is specific to C++ and cannot occur in C.

Other Languages

Languages in the Pascal/Ada family supports the nesting of function definitions. Such nested definitions essentially have static storage class. Java does not support the nesting of function definitions (although classes may be nested).

Common Implementations

gcc supports nested functions, as an extension, and the static storage-class can be explicitly specified for such functions. The usage is redundant in that they have block scope (inside the function that contains their definition) and are not visible outside of that block.

---

prior declaration not

If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage. 429

Commentary

A prior declaration might not be visible because it does not exist or because a block scope declaration of the same identifier (in the same name space) hides it. Here the term *no linkage* is used to mean that a linkage of external or internal was not assigned (using the rules of the standard to interpret a prior declaration).

outer scope 413
identifier hidden

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

Coding Guidelines

If a prior declaration exists, the situation involves more than one declaration of the same identifier. As such, it is covered by the guideline recommendation dealing with multiple declarations of the same identifier.

object 422
external linkage denotes same

Example

```
1  extern int doit_1(void); /* external linkage. */
2  static int doit_2(void); /* internal linkage. */
3
4  void f(void)
5  {
6  int doit_1, /* no linkage. */
7      doit_2; /* no linkage. */
8
9     {
10    extern int doit_1(void); /* external linkage. */
11    extern int doit_2(void); /* external linkage. */
12    }
13 }
```

430 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**.

function no storage-class

Commentary

A significant amount of existing code omits the **extern** specifier in function definitions. The Committee could not create a specification that required **extern**s to be inserted into long-existent source code. The behavior is as-if **extern** had appeared in the declaration. This does not automatically give the function external linkage (there could be a prior declaration that gives it internal linkage). This behavior is different from that used for object declarations in the same circumstances.

426 extern identifier linkage same as prior declaration
431 object file scope no storage-class

Other Languages

Languages that use some sort of specifier to denote the visibility of file scope identifiers usually apply the same rules to functions and objects.

Coding Guidelines

Developers generally believe that a function declaration that does not include a storage-class specifier has external linkage. While this belief is not always true, the consequences of it being wrong are not far-reaching (i.e., a program may fail to link because a call was added in a different source file to a function having internal linkage when the developer believed it had external linkage). There does not appear to be a worthwhile benefit in a guideline recommendation that changes the existing practice of not including **extern** in function definitions.

426 extern identifier linkage same as prior declaration

A future revision of the standard may require a storage-class specifier to appear in the declaration if a previous declaration of the identifier had internal linkage.

2035 identifier linkage future language directions

Example

```
1  static int f2(void);
2  int f2(void); /* Same behavior as-if extern int f2(void) had been written. */
3
4  extern int f(void)
5  { return 22; }
6
7  int g(void)
8  { return 23; }
```

431 If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

object file scope no storage-class

Commentary

function 430 no storage-class tentative definition 1849 tentative definition 1849 object 1354 reserve storage

There is no as-if here; the object is given external linkage. The behavior differs in more than one way from that for functions. If no storage-class specifier is given, the declaration is also a tentative definition. One possible reason for wanting to omit the storage-class specifier in the declaration of a file scope object is that such a declaration is also a definition (actually a tentative definition unless an explicit initializer is given). Another way of creating a definition at file scope with external linkage is to explicitly specify an initializer; in this case it does not matter if the storage-class specifier **extern** is or is not given.

Prior to C90, many implementations treated object declarations that contained both an **extern** storage-class specifier and an explicit initializer as a constraint violation. There was, and continues to be, a significant amount of existing code that omits the specification of any linkage on object declarations unless internal linkage is required. Without breaking existing code, the Committee continues to be in the position of not being able to change the specification, other than to make external linkage the default (when a storage-class specifier is not explicitly specified).

identifier 2035 linkage future language directions

Support for declaring an identifier with internal linkage at file scope without the storage-class specifier **static** may be withdrawn in a future revision of the standard.

Coding Guidelines

external declaration 1810 syntax

The issue of what linkage objects at file scope should be declared to have is discussed elsewhere.

Example

```
1  static int si1; /* internal linkage. */
2  int si1;        /* external linkage, previous was internal, linkage mismatch. */
3
4  int ei1;        /* No explicit storage-class given, so it is external.        */
5
6  extern int ei2; /* Same linkage as any prior declaration, otherwise external. */
```

---

identifier no linkage

The following identifiers have no linkage: 432

Commentary

no linkage 424 identifier declaration is unique

Here the phrase *no linkage* is used to mean that the linkage is *none*.

---

member no linkage

an identifier declared to be anything other than an object or a function; 433

Commentary

This list includes tags, typedefs, labels, macros, and a member of a structure, union, or enumeration.

C++

3.5p3 *A name having namespace scope (3.3.5) has internal linkage if it is the name of*

— a data member of an anonymous union.

While the C Standard does not support anonymous unions, some implementations support it as an extension.

3.5p4 *A name having namespace scope (3.3.5) has external linkage if it is the name of*

— a named class (clause 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.1.3); or

— a named enumeration (7.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.1.3); or

— an enumerator belonging to an enumeration with external linkage; or

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. 3.5p8

The following C definition may cause a link-time failure in C++. The names of the enumeration constants are not externally visible in C, but they are in C++. For instance, the identifiers `E1` or `E2` may be defined as externally visible objects or functions in a header that is not included by the source file containing this declaration.

```
extern enum T {E1, E2} glob;
```

There are also some C++ constructs that have no meaning in C, or would be constraint violations.

```
void f()
{
union {int a; char *p; }; /* not an object */
                          // an anonymous union object

/*
 * The following all have meaning in C++
 *
   a=1;
 *
   p="Derek";
 */
}
```

---

434 an identifier declared to be a function parameter; parameter linkage

Commentary

A parameter in a function definition is treated as a block scope object with automatic storage duration. A parameter in a function declaration has function prototype scope.

Other Languages

Some languages allow arguments to be passed to a function in any order. This is achieved by specifying the parameter identifier that a particular argument is being passed to at the point of call. The identifiers denoting the function parameters are thus visible outside of the function body. In such languages parameter identifiers can only occur in the context of an argument list to a call of the function that defines them.

Many languages do not permit identifiers to be specified for function declarations that are not also definitions. In these cases only the types of the parameters may be specified.

---

435 a block scope identifier for an object declared without the storage-class specifier **`extern`**. no linkage block scope object

Commentary

Block scope identifiers are only intended to be visible within the block that defines them. The linkage mechanism is not needed to resolve multiple declarations. Use of the storage-class specifier **`extern`** at block scope is needed to support existing (when the C90 was first created) code. Use of the storage-class specifier **`static`** at block scope creates an identifier with linkage none (the purpose of using the keyword in this context is to control storage duration, not linkage).

1645 identifier linkage at block scope
455 static storage duration

C++

3.5p8 *Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A name with no linkage (notably, the name of a class or enumeration declared in a local scope (3.3.2)) shall not be used to declare an entity with linkage.*

The following conforming C function is ill-formed in C++.

```
1  void f(void)
2  {
3  typedef int INT;
4
5  extern INT a; /* Strictly conforming */
6                // Ill-formed
7
8  enum E_TAG {E1, E2};
9
10 extern enum E_TAG b; /* Strictly conforming */
11                      // Ill-formed
12 }
```

Other Languages

Few languages provide any mechanism for associating declarations of objects in block scope with declarations in other translation units.

Coding Guidelines

identifier declared in one file 422.1

The guideline recommendation dealing with textually locating declarations in a header file is applicable here.

---

linkage both internal/external

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined. 436

Commentary

EXAMPLE linkage 1852

There are a few combinations of declarations where it is possible to give the same identifier both internal and external linkage. There are no combinations of declarations that can give the same identifier two other kinds of linkage.

C++

The C++ Standard does not specify that the behavior is undefined and gives an example (3.5p6) showing that the behavior is defined.

Common Implementations

Some translators issue a diagnostic for all cases where the same identifier appears with both internal and external linkage.

Coding Guidelines

An instance of this undefined behavior is most likely to occur when a `.c` file includes a header containing the declaration of an identifier for an object, where this `.c` file also contains a declaration using the storage-class specifier **`static`** for the same identifier (that is intended to be distinct from the identifier in the header). The situation occurs because the developer was not aware of all the identifiers declared in the header (one solution is to rename one of the identifiers). The same identifier is rarely declared with both internal and external linkage and a guideline recommendation is not considered worthwhile.

Example

```
1  extern int glob_0; /* external linkage. */
2  static int glob_0; /* internal linkage (prior declaration not considered). */
3
4  static int glob_1, glob_2; /* internal linkage. */
5         int         glob_2; /* Linkage is specified by the prior declaration. */
6
7  void f(void)
8  {
9  extern int glob_1; /* Same linkage as one visible at file scope. */
10
11        {
12        /*
13         * We need to enter another block to exhibit this problem.
14         * Now the visible declaration is in block scope, so the
15         * wording in clause 6.1.2.2 does not apply and we take the
16         * linkage from the declaration here, not the linkage from
17         * the outer scope declaration.
18         */
19         extern int glob_1; /* external and internal linkage. */
20         }
21 }
```

Usage

The translated form of this book's benchmark programs contained 27 instances of identifiers declared, within the same translation unit, with both internal and external linkage.

---

437 **Forward references:** declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

---

438 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. name space

Commentary

At a particular point in the source it is possible to have the same identifier declared and visible as a label, object/function/typedef/enumeration constant, enum/structure/union tag, and a potentially infinite number of member names. Syntactic context in C is much more localized than in human languages. For instance, in "There is a sewer near our home who makes terrific suites," the word *sewer* is not disambiguated until at least four words after it occurs. In C the token before an identifier (`->`, `.`, **`struct`**, **`union`**, and **`enum`**) disambiguates some name spaces. Identifiers used as labels require a little more context because the following `:` token can occur in several other contexts.

C++

This C statement is not always true in C++, where the name lookup rules can involve semantics as well as syntax; for instance, in some cases the **`struct`** can be omitted. 441 tag name space

Common Implementations

The amount of sufficient syntactic context needed to determine uses that refer to typedef names and other kinds of identifiers is greater than most implementors of translators want to use (for efficiency reasons most existing parser generators only use a lookahead of one token). Translators handle this by looking up identifiers in a symbol table prior to passing them on to be syntactically processed. For instance, the sequence `int f(a, b)` could be declaring `f` to be an old-style function definition (where `a` and `b` are the identifier names) or a function prototype (where `a` and `b` are the types of the parameters). Without accessing the symbol table, a translator would not know which of these cases applied until the token after the closing right parenthesis was seen.

Coding Guidelines

syntactic context
belief maintenance 0

Syntactic context is one source of information available to developers when reading source code. Another source of information are their existing beliefs (e.g., obtained from reading other source code related to what they are currently reading). Note: There is a syntactic context where knowledge of the first token does not provide any information on the identity a following identifier; a declaration defines a new identifier and the spelling of this identifier is not known until it is seen.

semantic priming 792

Many studies have found that people read a word (e.g., doctor) more quickly and accurately when it is preceded by a related word (e.g., nurse) than when it is preceded by an unrelated word (e.g., toast). This effect is known as *semantic priming*. These word pairs may be related because they belong to the same category, or because they are commonly associated with each other (often occurring in the same sentence together). Does the syntactic context of specific tokens (e.g., `->`) or keywords (e.g., **`struct`**) result in semantic priming? For such priming to occur there would need to be stronger associations in the developer's mind between identifiers that can occur in these contexts, and those that cannot. There have been no studies investigating how developers store and access identifiers based on their C language properties. It is not known if, for instance, an identifier appearing immediately after the keyword **`struct`** will be read more quickly and accurately if its status as a tag is known to the developer.

It is possible for identifiers denoting different entities, but with the same spelling, to appear closely together in the source. For instance, an identifier with spelling `kooncliff` might be declared as both a structure tag and an object of type **`float`** and appear in an expression in both contexts. Is syntactic context the dominant factor in such cases, or will the cost of comprehension be measurably greater than if the identifiers had different spellings?[438.1]

A study by McKoon and Ratcliff[920] looked at contextual aspects of meaning when applied to the same noun. Subjects were read a paragraph and then asked to reply *true* or *false* to three test questions (see the following discussion). Some of the questions involved properties that were implicit in the paragraph. For instance, an interest in painting a tomato implicitly involves its color. The results showed that subjects verified the property of a noun more quickly (1.27 seconds vs. 1.39) and with a lower error rate (4.7% vs. 9.7%), in a context in which the property was relevant than in a context in which it was not relevant.

> Paragraph 1
>
> This still life would require great accuracy. The painter searched many days to find the color most suited to use in the painting of the ripe tomato.
>
> Paragraph 2
>
> The child psychologist watched the infant baby play with her toys. The little girl found a tomato to roll across the floor with her nose.
>
> Test sentences
>
> 1) Tomatoes are red.
>
> 2) Balloons are heavy.
>
> 3) Tomatoes are round.

The results of this study show that the (category) context in which a name is used affects a subject's performance. However, until studies using C language constructs have been performed, we will not know the extent to which these context dependencies affect developer comprehension performance.

word frequency 792
identifier 792
tip-of-the-tongue

Are there situations where there might be a significant benefit in using identifiers with the same spelling, to denote different entities? Studies of word recall have found a frequency effect; the more concepts associated with a word, the fewer tip-of-the-tongue cases occur. Tag and typedef names denote very similar concepts, but they are both ways of denoting types.

[438.1] No faults can be introduced into the source through misuse of these identifiers because all such uses will result in a translator issuing a diagnostic.

```
1  typedef struct SOME_REC {int mem;} SOME_REC;
```

In the above case, any confusion on the developers' part will not result in unexpected affects (the name can appear either with or without the **struct** keyword— it is fail-safe).

Dev 792.3 A newly declared identifier denoting a tag may have the same spelling as a typedef name denoting the same type.

Macros provide a mechanism for hiding the syntactic context from readers of the visible source; for instance, the `offsetof` macro takes a tag name and a member name as arguments. This usage is rare and is not discussed further here.

Table 438.1: Identifiers appearing immediately to the right of the given token as a percentage of all instances of the given token. An identifier appearing to the left of a **:** could be a label or a **case** label. However, C syntax is designed to be parsed from left to right and the presence, or absence, of a **case** keyword indicates the entity denoted by an identifier. Based on the visible form of the `.c` files.

Token	.c file	.h file	Token	.c file	.h file
goto identifier	99.9	100.0	**struct** identifier	99.0	88.4
#define identifier	99.9	100.0	**union** identifier	65.5	75.8
. identifier	100.0	99.8	**enum** identifier	86.6	53.6
-> identifier	100.0	95.5	**case** identifier	71.3	47.2

439 Thus, there are separate *name spaces* for various categories of identifiers, as follows:

Commentary

In C each name space is separate from other name spaces. As the name suggests this concept can be viewed as a space within which names exist. This specification was introduced in C90 so that existing code was not broken. There is also a name space for macro names.

1928 macro one name space

The standard specifies various identifiers, having particular spellings, as being reserved for various purposes. The term *reserved name space* is sometimes used to denote identifiers belonging to this set. However, this term does not occur in the C Standard; it is an informal term used by the committee and some developers.

C++

C++ uses **namespace** as a keyword (there are 13 syntax rules associated with it and an associated keyword, **using**) and as such it is denotes a different concept from the C name space. C++ does contain some of the name space concepts present in C, and even uses the term namespace to describe them (which can be somewhat confusing). These are dealt with under the relevant sentences that follow.

Other Languages

Most languages have a single name space, which means that it is not necessary to talk about a name space as such. A few languages have a mechanism for separate translation that requires identifiers to be imported into a translation unit before they can be referenced. This mechanism is sometimes discussed in terms of being a name space.

Common Implementations

Some very early implementations of C did not always fully implement all of the name spaces defined in the C Standard. In particular early implementations treated the members of all structure and union types as belonging to the same name space[723] (i.e., once a member name was used in one structure type, it could not be used in another in the same scope).

Table 439.1: Occurrence of various kinds of declarations of identifiers as a percentage of all identifiers declared in all the given contexts. Based on the translated form of this book's benchmark programs.

Declaration Context	%	Declaration Context	%
block scope objects	23.7	file scope objects	4.4
macro definitions	19.3	macro parameters	4.3
function parameters	16.8	enumeration constants	2.1
struct/union members	9.6	**typedef** names	1.2
function declarations	8.6	tag names	1.0
function definitions	8.1	label names	0.9

label name space

— *label names* (disambiguated by the syntax of the label declaration and use); 440

Commentary

labeled statements 1722 syntax label name 394

Labels represent statement locations within a function body. Putting label names in their own name space represents a language design decision. It enables uses such as `if (panic) goto panic`.

C++

6.1p1 *Labels have their own name space and do not interfere with other identifiers.*

Other Languages

In Pascal and Ada labels must be declared before use. In Pascal they are numeric, not alphanumeric, character sequences. Some languages (e.g., Algol 60, Algol 68) do put label names in the same name space as ordinary identifiers.

Common Implementations

Some early implementations of C placed labels in the same name space as ordinary identifiers.

tag name space

— the *tags* of structures, unions, and enumerations (disambiguated by following any[24)] of the keywords **struct**, **union**, or **enum**); 441

Commentary

tag declarations 1459 different scope

The original definition of C[1170] did not include support for typedef names. Tags were the mechanism by which structure, union, and enumerated types could be given a name that could be referred to later. Putting tags in a different name space removed the possibility of a tag name clashing with an ordinary identifier. It also has the advantage on large projects of reducing the number of possible naming conflicts, and allowing declarations such as the following to be used:

```
1   typedef struct listitem listitem;
```

C++

Tags in C++ exist in what is sometimes known as *one and a half name spaces*. Like C they can follow the keywords **struct**, **union**, or **enum**. Under certain conditions, the C++ Standard allows these keywords to be omitted.

3.4p1 *The name lookup rules apply uniformly to all names (including `typedef-names` (7.1.3), `namespace-names` (7.3) and `class-names` (9.1)) wherever the grammar allows such names in the context discussed by a particular rule.*

In the following:

```
struct T {int i;};
struct S {int i;};
int T;

void f(T p); // Ill-formed, T is an int object
             /* Constraint violation */

void g(S p); // Well-formed, C++ allows the struct keyword to be omitted
             // There is only one S visible at this point
             /* Constraint violation */
```

C source code migrated to C++ will contain the **struct/union** keyword. C++ source code being migrated to C, which omits the *class-key*, will cause a diagnostic to be generated.

The C++ rules for tags and typedefs sharing the same identifier are different from C.

3.4.4p2

If the name in the elaborated-type-specifier is a simple identifier, and unless the elaborated-type-specifier has the following form:

class-key identifier ;

the identifier is looked up according to 3.4.1 but ignoring any non-type names that have been declared. If this name lookup finds a typedef-name, the elaborated-type-specifier is ill-formed.

The following illustrates how a conforming C and C++ program can generate different results:

```
extern int T;

int size(void)
{
struct T {
         double mem;
         };

return sizeof(T);   /* sizeof(int) */
                    // sizeof(struct T)
}
```

The following example illustrates a case where conforming C source is ill-formed C++.

```
struct TAG {int i;};
typedef float TAG;

struct TAG x; /* does not affect the conformance status of the program */
              // Ill-formed
```

Other Languages

Tags are unique to C (and C++).

Coding Guidelines

A tag name always appears to the right of a keyword. Its status as a tag is clearly visible to the reader. The issue of tag naming conventions is discussed elsewhere.

792 tag naming conventions

Example

```
typedef struct X_REC {
                      int mem1;
                      } X_REC;
```

members name space

— the *members* of structures or unions; 442

Commentary

base document 1

In the base document, all members of structure and union types occupied the same name space (an idea that came from BCPL). Such member names were essentially treated as symbolic forms for offsets into objects.

```
1  struct {
2          char mem_1;
3          long mem_2;
4         } x;
5  struct {
6         char mem_3;
7         char mem_4;
8         } y;
9
10 void f(void)
11 {
12 /*
13  * Under the original definition of C the following would zero a char
14  * sized object in x that had the same offset as mem_4 had in y.
15  */
16 x.mem_4 = 0;
17 }
```

This language specification prevented structures and unions from containing a particular member name once it had been used in a previous definition as a member name. The idea of member names representing offsets was a poor one and quickly changed. A consequence of this history is that structure and union definitions are thought of as creating a name space, not in terms of members existing in a scope (a usage common to many other languages, including C++).

C++

3.3.6p1 *The following rules describe the scope of names declared in classes.*

In C++ members exist in a scope, not a name space.

```
1  struct {
2         enum E_TAG { E1, E2}  /* C identifiers have file scope */
3                               // C++ identifiers have class scope
4                               m1;
5         } x;
6
7  enum E_TAG y; /* C conforming */
8                // C++ no identifier names E_TAG is visible here
```

Other Languages

Many languages treat the contents of what C calls a structure or union as a scope. The member-selection operator opening this scope to make the members visible (in C this operator makes the unique name space visible).

member namespace

each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . or -> operator); 443

Commentary

Although each structure or union type definition creates its own unique name space, no new scope is created. The declarations contained within these definitions have the scope that exists at the point where the structure or union is defined.

406 identifier scope determined by declaration placement

C++

The name of a class member shall only be used as follows: 3.3.6p2

. . .

— *after the* `.` *operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,*

— *after the* `->` *operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,*

```
1  struct {
2         enum {E1, E2} m;
3         } x;
4
5  x.m = E1; /* does not affect the conformance status of the program */
6            // ill-formed. X::E1 is conforming C++ but a syntax violation in C
```

Coding Guidelines

Given that one of two tokens immediately precedes a member name its status as a member is immediately obvious to readers of the source (the only time when this context may not be available is when a member name occurs as an argument in a macro invocation). Because of the immediate availability of this information there is no benefit in a naming convention intended to flag the status of an identifier as a member.

Members in different structure types may hold the same kind of information; for instance, a member named `next` might always point to the next element in a linked list, its type being a pointer to the structure type containing its definition. Members named `x_coord` and `y_coord` might occur in several structure types dealing with coordinate systems.

The underlying rationale for the guideline recommendation dealing with reusing the same identifier name, confusion cased by different semantic associations, is only applicable if the semantic associations are different. If they are the same, then there is a benefit in reusing the same name when dealing with different members that are used for the same purpose.

792.3 identifier reusing names

Dev 792.3

A newly declared member may have the same spelling as a member in a different structure/union type provided they both share the same semantic associations; and if they both have an arithmetic type, their types are the same.

The term *same semantic associations* is somewhat ill-defined and subject to interpretation (based on the cultural background and education of the person performing the evaluation, an issue that is discussed in more detail elsewhere). While guideline recommendation wording should normally aim to be precise, it should also try to be concise. It is not clear that this guideline would be improved by containing more words.

792 semantic associations enumerating

Members having arithmetic types could be interchanged in many situations without any diagnostic being issued. A member having type **`int`** in one structure and type **`float`** in another, for instance, may represent the same semantic concept; but the way they need to be handled is different. Inappropriately interchanging nonarithmetic types is much more likely to result in a diagnostic being generated.

A study by Neamtiu, Foster, and Hicks[998] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 79% of releases one or more existing structure

or union types had one or more fields added to them, while structure or union types had one or more fields deleted in 51% of releases and had one or more of their field names changed in 37% of releases. One or more existing fields had their types changed in 35% of releases.[997]

A study by Anquetil and Lethbridge[47] analyzed 2 million lines of Pascal (see Table 443.1 and Table 443.2). Members that shared the same name were found to be much more likely to share the same type than members having different names.

Table 443.1: Number of matches found when comparing between pairs of members contained in different Pascal records that were defined with the same type name. Adapted from Anquetil and Lethbridge.[47]

	Member Types the Same	Member Types Different	Total
Member names the same	73 (94.8%)	4 (5.2%)	77
Member names different	52 (11 %)	421 (89 %)	473

Table 443.2: Number of matches found when comparing between pairs of members contained in different Pascal records (that were defined with the any type name). Adapted from Anquetil and Lethbridge.[47]

	Member Types the Same	Member Types Different	Total
Member names the same	7,709 (33.7%)	15,174 (66.3%)	22,883
Member names different	158,828 (0.2%)	66,652,062 (99.8%)	66,710,890

Example

```
1  struct {
2          int m;
3         } x;
4  struct {
5          int m;
6          struct {
7                  int n;
8                 } n;
9         } y;
```

the two members, named `m`, are each in a different name space. In the definition of `y` the two members, named `n`, are also in different name spaces.

ordinary identifiers
name space ordinary identifiers

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants). 444

Commentary

This defines the term *ordinary identifiers*. These ordinary identifiers include objects, functions, typedef names and enumeration constants. Macro names and macro parameters are not included in this list. Keywords are part of the language syntax. Identifiers with the spelling of a keyword only stop being identifier preprocessing tokens and become keyword tokens in phase 7. Literals are not in any name space.

C++

The C++ Standard does not define the term *ordinary identifiers*, or another term similar to it.

Coding Guidelines

Naming conventions that might be adopted to distinguish between these ordinary identifiers, used for different purposes, are discussed in more detail in their respective sentences.

enumeration 792 constant naming conventions
typedef 792 naming conventions
macro 792 naming conventions
function 792 naming conventions

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the **goto** statement (6.8.6.1). 445

446 23) As specified in 6.2.1, the later declaration might hide the prior declaration. footnote 23

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  int glob; /* Hide prior declaration. */
6     {
7     extern int glob;
8     }
9  }
```

447 24) There is only one name space for tags even though three are possible. footnote 24

Commentary

The additional functionality, if three had been specified, would be to enable the same identifier spelling to be used after each of the three keywords. There is little to be gained from this and the possibility of much confusion.

C++

There is no separate name space for tags in C++. They exist in the same name space as object/function/typedef *ordinary identifiers*.

Common Implementations

Early translators allowed the **`struct`** and **`union`** keywords to be intermixed.

```
1  union Tag {
2            int mem;
3            };
4
5  int main(void)
6  {
7  struct Tag *ptr;   /* Acceptable in K&R C. */
8  }
```

6.2.4 Storage durations of objects

448 An object has a *storage duration* that determines its lifetime. storage duration object

Commentary

Storage duration is a property unique to objects. In many cases it mirrors an object's scope (not its visibility) and developers sometimes use the term *scope* when *lifetime* would have been the correct term to use.

C++

An object has a storage duration (3.7) which influences its lifetime (3.8). 1.8p1

In C++ the initialization and destruction of many objects is handled automatically and in an undefined order (exceptions can alter the lifetime of an object, compared to how it might appear in the visible source code). For these reasons an object's storage duration does not fully determine its lifetime, it only influences it.

Other Languages

Most languages include the concept of the lifetime for an object.

There are three storage durations: static, automatic, and allocated. 449

Commentary

One of the uses of the overworked keyword **`static`** is to denote objects that have static storage duration (there are other ways of denoting this storage duration).

C90

The term *allocated* storage duration did not appear in the C90 Standard. It was added by the response to DR #138.

C++

3.7p1 *The storage duration is determined by the construct used to create the object and is one of the following:*

— *static storage duration*

— *automatic storage duration*

— *dynamic storage duration*

The C++ term *dynamic storage* is commonly used to describe the term *allocated storage*, which was introduced in C99.

Common Implementations

stack

Objects having particular storage durations are usually held in different areas of the host address space. The amount of static storage duration is known at program startup, while the amounts of automatic and allocated storage varies during program execution. The most commonly seen division of available storage is to have the two variable-size storage durations growing toward each other. Objects having static storage duration are often located at the lowest address rather than the highest. This design decision may make it possible to access these objects with a single instruction using a register + offset addressing mode (provided one is available).

register + offset 1000

A few implementations do not have separate stack and heap areas. They allocate stack space on the heap, on an as-needed basis. This usage is particularly common in realtime, multiprocess environments, without hardware memory management support to map logical addresses to different physical addresses. The term *cactus stacks* is sometimes used.

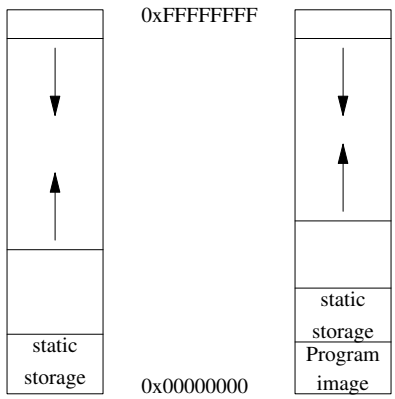


Figure 449.1: The location of the stack invariably depends on the effect of a processor's pop/push instructions (if they exist). The heap usually goes at the opposite end of available storage. The program image may, or may not, exist in the same address space.

Some processors (usually those targeted at embedded systems[612]) support a variety of different kinds of addressable storage. This storage may be disjoint in that two storage locations can have the same address, accesses to them being disambiguated either by the instruction used or a flag specifying the currently active storage bank. Optimally allocating declared objects to these storage areas is an active research topic.[66, 1250] One implementation[66] distributes the stack over several storage banks.

Coding Guidelines

In resource-constrained environments there can be space and efficiency issues associated with the different kinds of storage durations. These are discussed for each of the storage durations later.

The term *allocated storage* is not commonly used by developers (in the sense of being a noun). The use of the word *allocated* as an adjective is commonly heard. The terms *dynamically allocated* or *allocated on the heap* are commonly used to denote this kind of storage duration. There does not seem to be any worthwhile benefit in trying to educate developers to use the technically correct term in this case.

Usage

In the translated form of this book's benchmark programs 37% of defined objects had static storage duration and 63% had automatic storage duration (objects with allocated storage duration were not included in this count).

Table 449.1: Total number of objects allocated (in thousands), the total amount of storage they occupy (in thousands of bytes), their average size (in bytes) and the high water mark of these values (also in thousands). Adapted from Detlefs, Dosser and Zorn.[349]

Program	Total Objects	Total Bytes	Average Size	Maximum Objects	Maximum Bytes
sis	63,395	15,797,173	249.2	48.5	1,932.2
perl	1,604	34,089	21.3	2.3	116.4
xfig	25	1,852	72.7	19.8	1,129.3
ghost	924	89,782	97.2	26.5	2,129.0
make	23	539	23.0	10.4	208.1
espresso	1,675	107,062	63.9	4.4	280.1
ptc	103	2,386	23.2	102.7	2,385.8
gawk	1,704	67,559	39.6	1.6	41.0
cfrac	522	8001	15.3	1.5	21.4

450 Allocated storage is described in 7.20.3.

Commentary

Allocated storage is not implicitly handled by the implementation. It is controlled by calling library functions.

Other Languages

In some languages handle allocated storage is part of the language, not the library. For instance, C++ contains the **new** operator (where the amount of storage to allocate is calculated by the translator, based on deducing the type of object required). Pascal also contains new, but calls is a *required function*.

451 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it.

lifetime of object

Commentary

This defines the term *lifetime*. The storage reserved for an object may exist outside of its guaranteed lifetime. However, this behavior is specific to an implementation and cannot be relied on in a program.

C90

The term *lifetime* was used twice in the C90 Standard, but was not defined by it.

C++

3.8p1 *The lifetime of an object is a runtime property of the object. The lifetime of an object of type T begins when:*

. . .

The lifetime of an object of type T ends when:

The following implies that storage is allocated for an object during its lifetime:

3.7p1 *Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object.*

Common Implementations

An implementation may be able to deduce that it is only necessary to reserve storage for an object during a subset of the lifetime required by the standard. For instance, in the following example the accesses to the objects `loc_1` and `loc_2` occur in disjoint portions of program execution. This usage creates an optimization opportunity (having the two objects share the same storage during disjoint intervals of their lifetime).

```
1  void f(void)
2  {
3  int loc_1,
4      loc_2;
5  /*
6   * Portion of program execution that accesses loc_1, but not loc_2.
7   */
8
9  /*
10  * Portion of program execution that accesses loc_2, but not loc_1.
11  */
12 }
```

Reuse of storage is usually preceded by some event; for instance, a function return followed by a call to another function (for automatic storage duration, the storage used by the previous function is likely to be used by different objects in the newly called function). There are many possible algorithms[9, 17, 96, 375, 484, 1144, 1174] that an implementation can use to manage allocated storage and no firm prediction on reuse can be made about objects having this storage duration.

A study by Bowman, Ratliff, and Whalley[148] optimized the storage lifetimes of the static data used by 15 Unix utilities (using the same storage locations for objects accessed during different time intervals; they all had static storage duration). They were able to make an overage saving of 7.4%. By overlaying data storage with instruction storage (the storage for instructions not being required after their last execution) they achieved an average storage saving of 22.8%.

object 1354
reserve storage

The issue of where storage is allocated for objects is discussed elsewhere.

Coding Guidelines

The lifetime of objects having static or automatic storage durations are easy to deduce from looking at the source code. The lifetime of allocated storage is rarely easy to deduce. Coding techniques to make it easier to demark the lifetime of allocated storage are outside the scope of this book (the Pascal `mark`/`release` functionality sometimes encouraged developers to develop algorithms to treat heap allocation in a stack-like fashion).

Example

The following coding fault is regularly seen:

```
1  struct list *p;
2    /* ... */
3  free(p);
4  p = p->next;
```

452 An object exists, has a constant address,[25)] and retains its last-stored value throughout its lifetime.[26)]

Commentary

At first glance the phrase *constant address* appears to prevent a C implementation from using a garbage collector that moves objects in storage. But what is an address? An implementation could choose to represent object addresses as an index into an area of preallocated storage. This indexed element holds the real address in memory of the object's representation bits. The details of this extra indirection operation is dealt with by the translator (invisible to the developer unless a disassembled listing of the generated code is examined). A garbage collector would only need to update the indexed elements after storage had been compacted, and the program would know nothing about what had happened.

The last-stored value may have occurred as a result of an assignment operator, external factors for an object declared with the **volatile** storage-class, or another operator that updates the value held in an object.

C++

There is no requirement specified in the C++ Standard for an object to have a constant address. The requirements that are specified include:

> *Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).* 1.9p10

> *All objects which neither have dynamic storage duration nor are local have static storage duration. The storage for these objects shall last for the duration of the program (3.6.2, 3.6.3).* 3.7.1p1

Common Implementations

In most implementations objects exist in either RAM or ROM. The value of an object whose address is never taken may only ever exist in a processor register; the only way to tell is by looking at a listing of the generated machine code. In all commercial implementations known to your author an object's address has a direct correspondence to its actual address in storage. There is no indirection via other storage performed by the implementation, although the processor's memory-management unit may perform its own mappings into physical memory.

Coding Guidelines

An implementation that performs garbage collection may have one characteristic that is visible to the developer. The program may appear to stop executing periodically because garbage collection is taking place. There are implementation techniques that perform incremental garbage collection, which avoids this problem to some degree.[689] However, this problem is sufficiently rare that it is considered to be outside the scope of these coding guidelines.

A program running in a sufficiently hostile environment that the last-stored value of an object may be corrupted requires the use of software development techniques that are outside the scope of this book.

Example

The two values output on each iteration of the loop are required to be equal. However, there is no requirement that the values be the same for different iterations of the loop.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  for (int loop=0; loop < 10; loop++)
6     {
7     int nested_obj;
8
9     printf("address is=%p", &nested_obj);
10    printf(", and now it is=%p\n", &nested_obj);
11    }
12 }
```

If an object is referred to outside of its lifetime, the behavior is undefined. 453

Commentary

Such a reference is possible through

- the address of a block scope object assigned to a pointer having a greater lifetime, and
- an object allocated by the memory-allocation library functions that has been freed.

C++

The C++ Standard does not unconditionally specify that the behavior is undefined (the cases associated with pointers are discussed in the following C sentence):

3.8p3 *The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [Note: in particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.6.2 and in 12.7. describe the behavior of objects during the construction and destruction phases.]*

Common Implementations

object 458
lifetime from entry to exit of block

On entry to a function it is common for the total amount of stack storage for that invocation to be allocated. The extent to which the storage allocated to objects, defined in nested blocks, is reused will depend on whether their lifetime is disjoint from objects defined in other nested blocks. In other words, is there an opportunity for a translator to reuse the same storage for different objects?

Most implementations' undefined behavior is to continue to treat the object as if it was still live. The storage location referenced may contain a different object, or even some internal information used by the runtime system. Values read from that location may be different from the last-stored value written into the original object. Stores to that location could affect the values of other objects and the effects of modifying internal, housekeeping information can cause a program to abort abnormally. The situation with allocated storage is much more complex.

Coding Guidelines

object reference outside lifetime

When the lifetime of an object ends, nothing usually happens to the last-stored value (or indeed any subsequent ones) held at that location. Programs that access the storage location that held the object, soon after the object's lifetime has ended, often work as expected (such an access can only occur via a pointer dereference; if an identifier denoting a declared object is visible the object it denotes cannot be outside of its lifetime). Accessing an object outside of its lifetime is unlikely to cause the implementation to issue a diagnostic. However, accessing an object outside of its lifetime sometimes does result in unexpected behavior. A coding guideline recommending that "an object shall not be referenced outside of its lifetime." is a special case of the guideline stating that programs shall not contain faults. The general aim of guideline recommendations in

guidelines 0
not faults

this area is to prevent the address of an object being available outside of its lifetime.

Although some implementations provide a mechanism to initialize newly created objects with some unusual value (this often helps to catch uninitialized objects quickly during testing), an equivalent mechanism at the end of an object's lifetime is unknown (to your author).

---

454 The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

pointer indeterminate

Commentary

It is not the object pointed at, but the value of pointers pointing at it that become indeterminate. Once its value becomes indeterminate, the value of the pointer cannot even be read; for instance, compared for equality with another pointer.

An object having a pointer type has an indeterminate value at the start of its lifetime, like any other object (even if that lifetime starts immediately after it was terminated; for instance, an object defined in the block scope of a loop).

461 object initial value indeterminate

C++

The C++ Standard is less restrictive; if does not specify that the value of the pointer becomes indeterminate.

> *Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[34)] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.3.2), and using the pointer as if the pointer were of type* `void*`*, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. If the object will be or was of a class type with a nontrivial destructor, and the pointer is used as the operand of a* `delete-expression`*, the program has undefined behavior.*

3.8p5

Source developed using a C++ translator may contain pointer accesses that will cause undefined behavior when a program image created by a C implementation is executed.

Other Languages

Languages in the Pascal/Ada family only allow pointers to refer to objects with allocated storage lifetime. These objects can have their storage freed. In this case the same issues as those in C apply.

Common Implementations

Some processors load addresses into special registers (sometimes called address registers; for instance, the Motorola 68000[968]). Loading a value into such an address register may cause checks on its validity as an address to be made by the processor. If the referenced address refers to storage that is no longer available to the program, a trap may be generated.

pointer cause undefined behavior

Coding Guidelines

One way to ensure that pointers never refer to objects whose lifetime has ended is to ensure they are never assigned the address of an object whose lifetime is greater than their own. Scope is a concept that developers are more familiar with than lifetime and a guideline recommendation based on scope is likely to be easier to learn. The applicable recommendations is given elsewhere.

1088.1 object address assigned

Returning the address of a block scope object, in a **`return`** statement, is a fault. Although other guidelines sometimes recommend against this usage, these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

0 guidelines not faults

---

455 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*.

static storage duration

Commentary

This defines the term *static storage duration*. Objects have storage duration, identifiers have linkage. The visibility of an identifier defined with internal linkage may be limited to the block that defined it, but its lifetime extends from program startup to program termination.

Objects declared in block scope can have static storage duration. The **extern** or **static** storage-class specifiers can occur on declarations in block scope. In the former case it refers to a definition outside of the block. In the latter case it is the definition.

string literal 903
static storage duration

All file scope objects have static storage duration. String literals also have static storage duration.

C++

The wording in the C++ Standard is not based on linkage and corresponds in many ways to how C developers often deduce the storage duration of objects.

3.7.1p1 *All objects which neither have dynamic storage duration nor are local have static storage duration.*

3.7.1p3 *The keyword* **static** *can be used to declare a local variable with static storage duration.*

Common Implementations

The total number of bytes of storage required for static storage duration objects is usually written into the program image in translation phase 8. During program startup this amount of storage is requested from the host environment. Objects and literals that have static storage duration are usually placed in a fixed area of memory, which is reserved on program startup. This is possible because the amount of storage needed is known prior to program execution and does not change during execution.

Coding Guidelines

There can be some confusion, in developers' minds, between the keyword **static**, static storage duration, and *declared static*— a term commonly used to denote internal linkage. There are even more uses of the keyword **static** discussed elsewhere. There is little to be done, apart from being as precise as possible, to reduce the number of possible alternatives when using the word static.

declarator syntax 1547

Objects in block scope, having static storage duration, retain their last-stored value between invocations of the function that contains their definition. Mistakes are sometimes made on initializing such objects; the developer forgets that initialization via a statement, rather than via an initialization expression in the definition, will be executed every time the function is called. Assigning a value to such an object as its first usage suggests that either static storage duration was not necessary or that there is a fault in the code. While this usage might be flagged by a lint-like tool, neither of them fall within the remit of guideline recommendations.

Example

Here are three different objects, all with static storage duration.

```
1  static int vallu;
2
3  void f(void)
4  {
5  static int valyou = 99;
6
7     {
8     static int valu;
9
10    valu = 21;
11    }
12 }
```

Usage

In the visible form of the `.c` files approximately 5% of occurrences of the keyword **static** occurred in block scope.

456 Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

static storage duration when initialized

Commentary

The storage is allocated and initialized prior to calling the function `main`. A recursive call to `main` does not cause startup initialization to be performed again.

150 program startup

Other Languages

Java loads modules on an as-needed basis. File scope objects only need be allocated storage when each module is loaded, which may be long after the program execution started.

Common Implementations

Implementations could use the as-if rule to delay creating storage for an object, with static storage duration, until the point of its first access. There were several development environments in the 1980s that used incremental linkage at the translation unit level. These environments were designed to aid the testing and debugging of programs, even if the entire source base was not available.

Coding Guidelines

In environments where storage is limited, developers want to minimize the storage footprint of a program. If some objects with static storage duration are only used during part of a program's execution, more efficient storage utilization schemes may be available; in particular making use of allocated storage.

Use of named objects makes it easier for a translator, or static analysis tool, to detect possible defects in the source code. Use of pointers to objects requires very sophisticated points to analysis just to be able to do the checks performed for named objects (without using sophisticated analysis).

A technique that uses macros to switch between referencing named objects during development and allocated storage during testing and deployment offers a degree of independent checking. If this technique is used, it is important that testing be carried out using the configuration that will ship in the final product. Using named objects does not help with checking the lifetimes of the allocated objects used to replace them. However, discussion of techniques for controlling a program's use of storage is outside the scope of this book.

457 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*.

automatic storage duration

Commentary

These objects occur in block scope and are commonly known as *local variables*. The storage-class specifier **`auto`** can also be used in the definition of such objects. Apart from translator test programs, this keyword is rarely used, although some developers use it as a source layout aid.

C++

3.7.2p1

> *Local objects explicitly declared* **`auto`** *or* **`register`** *or not explicitly declared* **`static`** *or* **`extern`** *have automatic storage duration.*

Other Languages

Nearly all languages have some form of automatic storage allocation for objects. Use of a language keyword to indicate this kind of storage allocation is very rare. Cobol does not support any form of automatic storage allocation.

Common Implementations

Storage for objects with automatic storage duration is normally reserved on a stack. This stack is frequently the same one used to pass arguments to functions and to store function return addresses and other housekeeping information associated with a function invocation (see Figure 1000.1).

Recognizing the frequency with which such automatic storage duration objects are accessed (at least while executing within the function that defines them), many processor instruction sets have special operations, or addressing modes, for accessing storage within a short offset from an address held in a register.

register 1000 + offset

The Dynamic C implementation for Rabbit Semiconductor[1499] effectively assigns the static storage class to all objects declared with no linkage (this default behavior, which does not support recursive function calls, may be changed using the compiler **`#class auto`** directive).

Coding Guidelines

The terminology *automatic storage duration* is rarely used by developers. Common practice is to use the term *block scope* to refer to such objects. The special case of block scope objects having static storage duration is called out in the infrequent cases it occurs.

---

object lifetime from entry to exit of block

For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. 458

Commentary

While the lifetime of an object may start on entry into a block, its scope does not start until the completion of the declarator that defines it (an objects scope ends at the same place as its lifetime).

identifier 418 scope begins
block scope 408 terminates

C++

3.7.2p1 *The storage for these objects lasts until the block in which they are created exits.*

5.2.2p4 *The lifetime of a parameter ends when the function in which it is defined returns.*

6.7p2 *Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).*

Which is a different way of phrasing 3.7.2p1.

3.8p1 *The lifetime of an object of type T begins when:*

— storage with the proper alignment and size for type T is obtained, and

. . .

The lifetime of an object of type T ends when:

— the storage which the object occupies is reused or released.

The C++ Standard does not appear to completely specify when the lifetime of objects created on entry into a block begins.

Other Languages

All arrays in Java, Awk, Perl, and Snobol 4 have their length decided during program execution (in the sense that the specified size is not used during translation even if it is a constant expression). As such, the lifetime of array in these languages does not start until the declaration, or statement, that contains them is executed.

Common Implementations

Most implementations allocate the maximum amount of storage required, taking into account definitions within nested blocks, on function entry. Such a strategy reduces to zero any storage allocation overhead associated with entering and leaving a block, since it is known that once the function is entered the storage requirements for all the blocks it contains are satisfied. Allocating and deallocating storage on a block-by-block basis is usually considered an unnecessary overhead and is rarely implemented. In:

```
void f(void)
{               /* block 1 */
int loc_1;

   {            /* block 2 */
   long loc_2;
   }

   {            /* block 3 */
   double loc_3;
   }
}
```

the storage allocated for `loc_2` and `loc_3` is likely to overlap. This is possible because the blocks containing their definitions are not nested within each other, and their lifetimes are disjoint. On entry to `f` the total amount of storage required for developer-defined automatic objects (assuming `sizeof(long) <= sizeof(double)`) is `sizeof(int)+padding_for_double_alignment+sizeof(double)`.

Coding Guidelines

Where storage is limited, defining objects in the closest surrounding block containing accesses to them, can reduce the maximum storage requirements (because objects defined with disjoint lifetimes can share storage space). It is safer to let the translator perform the housekeeping needed to handle such shared storage than to try to do it manually (by using of the same objects for disjoint purposes). The issues surrounding the uses to which an object is put are discussed elsewhere. The issues involved in deciding which block an identifier should be defined in, if it is only referenced within a nested block, are also discussed elsewhere.

1352.1 object used in a single role
1348 identifier definition close to usage

Example

```
int glob;
int *pi = &glob;

void f(void)
{ /* Lifetime of loc starts here. */

block_start:;

*pi++; /* The identifier loc is not visible here. */

if (*pi == 1)
   goto skip_definition; /* Otherwise we always execute initializer. */

int loc = 1;

skip_definition:;

pi=&loc;

if (loc != 7)
   goto block_start;

/* Lifetime of loc ends here. */ }
```

(Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) 459

Commentary

Execution of a block ends when control flow within that block transfers execution to a block at the same or lesser block nesting, or causes execution of the function in which it is contained to terminate (i.e., by executing a **return** statement, or calling one of the `longjmp`, `exit`, or `abort` functions).

C is said to be a *block structured* language.

C++

1.9p10 *Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).*

3.7.2p1 *The storage for these objects lasts until the block in which they are created exits.*

Other Languages

This behavior is common to block structured languages.

Coding Guidelines

It would be incorrect to assume that objects defined with the **volatile** qualifier can only be modified by the implementation while the block that defines them is being executed. Such objects can be modified at any point in their lifetime.

---

block entered recursively

If the block is entered recursively, a new instance of the object is created each time. 460

Commentary

function call 1026 recursive

This can only happen through a recursive call to the function containing the block. A jump back to the beginning of the block, using a **goto** statement, is not a recursive invocation of that block.

C90

The C90 Standard did not point this fact out.

C++

object 458 lifetime from entry to exit of block

As pointed out elsewhere, the C++ Standard does not explicitly specify when storage for such objects is created. However, recursive instances of block scope declarations are supported.

5.2.2p9 *Recursive calls are permitted, except to the function named `main` (3.6.1).*

6.7p2 *Variables with automatic storage duration (3.7.2) are initialized each time their `declaration-statement` is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).*

Other Languages

This behavior is common to block structured languages. Recursion was not required in the earlier versions of the Fortran Standard, although some implementations provided it.

Common Implementations

There are some freestanding implementations where storage is limited and recursive function calls are not supported (such implementations are not conforming). The problem is not usually one of code generation, but the storage optimizations performed by the linker. To minimize storage usage in memory limited environments linkers build program call graphs to deduce which objects can have their storage overlaid. The storage optimization algorithms do not terminate if there are loops in the call graph (a recursive call would create such a loop). Thus, recursion is not supported because programs containing it would never get past translation phase 8.

1000 call graph

Allocating storage for all objects defined in a function, on a per function invocation basis, may cause inefficient use of storage when recursive invocations occur. In practice both recursive invocations and objects defined in nested blocks are rare.

---

461 The initial value of the object is indeterminate.

object initial value indeterminate

Commentary

This statement applies to all object declarations having no linkage, whether they have initializers or not (it is explicitly stated for objects having a VLA type elsewhere).

466 object initial VLA value indeterminate

C++

8.5p9

If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor.

Otherwise, if no initializer is specified for an object, the object and its subobjects, if any, have an indeterminate initial value[90]; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

C does not have constructors. So a const-qualified object definition, with a structure or union type, would be ill-formed in C++.

```
1  struct T {int i;};
2
3  void f(void)
4  {
5  const struct T loc; /* very suspicious, but conforming */
6                      // Ill-formed
7  }
```

Other Languages

This behavior is common to nearly every block scoped language. Some languages (e.g., awk) provide an initial value for all objects (usually zero, or the space character).

Common Implementations

Some implementations assign a zero value to automatic and allocated storage when it is created. They do this to increase the likelihood that programs containing accesses to uninitialized objects will work as intended. They are *application user friendly* by helping to protect against errors in the source code. (Many objects are initially set to zero by developers and this implicit implementation value assignment mimics what is likely to be the behavior intended by the author of the code.)

Some implementations (e.g., the Diab Data compiler[353] supports the `-Xinit-locals-mask` option) implicitly assign some large value, or a trap representation, to freshly allocated storage. The intent is to be *developer friendly* by helping to detect faults, created by use of uninitialized objects, as quickly as possible. Assigning an unusual value is likely to have the effect of causing the reads from uninitialized objects to

have an unintended effect and to be quickly detected. The Burroughs B6700 (a precursor of the Unisys A Series[1389]) initialized its 48-bit words to 0xbadbadbadbad. The value 0xdeadbeef is used in a number of environments supported by IBM.

Coding Guidelines

It is surprising how often uninitialized objects contain values that result in program execution producing reasonable results. Having the implementation implicitly initialize objects to some *unfriendly* value helps to track down these kinds of faults much more quickly and helps to prevent reasonable results from hiding latent problems in the source code.

---

initialization performed every time declaration reached

If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; 462

Commentary

object 1711 initializer evaluated when

The sequence of operations is specified in more detail elsewhere in the C Standard.

If flow of control jumps over the declaration, for instance by use of a **goto** statement, the initialization is not performed; however, the lifetime of the object is not affected (the storage will already have been allocated on entry into the block).

object 458 lifetime from entry to exit of block

Because initializations are performed during program execution, it is possible for the evaluation of a floating constant to cause an exception to be raised. This can occur even if the floating constant appears to have the same type as the object it is being assigned to. An implementation is at liberty to evaluate the initialization expression in a wider format, which will then need to be converted to the object type. It is the conversion from any wider format to the type of the object type that may raise the exception.

FLT_EVAL_METHOD 354

```
1  #include <math.h>
2
3  void f(void)
4  {
5  static float w = 1.1e75; /* Performed at translation time, no exception raised. */
6  /*
7   * The following may all require conversions at execution time.
8   * Therefore they can all raise exceptions.
9   */
10 float x_1 = 1.1e75;
11 double x_2 = 1.1e75;
12 float x_3 = 1.1e75f;
13
14 /*
15  * The following do not require any narrowing conversions and cannot raise exceptions.
16  */
17 long double x_4 = 1.1e75;
18 double_t x_5 = 1.1e75;
19
20 /*
21  * For constant expressions we have (the final value of y and z is undefined)...
22  */
23 static double y = 0.0/0.0;  /* Performed at translation time, no exception raised. */
24 auto double z = 0.0/0.0;    /* Performed at execution time, may raise exception.   */
25 }
```

C90

If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement.

Support for mixing statements and declarations is new in C99. The change in wording is designed to ensure that the semantics of existing C90 programs is unchanged by this enhanced functionality.

Other Languages

Some languages do not allow object definitions to contain an initializer. Those that do usually follow the same initialization rules as C.

Common Implementations

For objects having a scalar type the machine code generated for the initializer will probably look identical to that generated for an assignment statement. For derived types implementations have all the information needed to generate higher-quality code. For instance, for array types, the case of all elements having a zero value is usually a special case and a machine code loop is generated to handle it.

Example

```
1  void f(int x)
2  {
3  switch(x)
4     {
5     int i=33; /* This initialization never occurs. */
6
7     case 1: x--;
8             break;
9     }
10 }
```

Usage

Usage information on initializers is given elsewhere.

1652 object value indeterminate

---

463 otherwise, the value becomes indeterminate each time the declaration is reached.

object indeterminate each time declaration reached
1711 object initializer evaluated when

Commentary

An indeterminate value is stored in the object each time the declaration is reached in the order of execution.

Other Languages

Most languages follow the model of giving an object an indeterminate value when it is first defined.

Common Implementations

If no other objects are assigned the same storage location, it is very likely that the last-stored value will be available in the object every time its declaration is reached.

Example

```
1  extern int glob;
2
3  _Bool f(void)
4  {
5  for (int i=0; i<5; i++)
6     {
7     int loc;
8
9     if (i > 0)
10       loc -= i; /* loc always has an indeterminate value here. */
11    else
12       loc=23;
13    }
14
15 goto do_work;
16    {
17    start_block:;
```

```
18      int count;
19
20      return (count == glob); /* count always has an indeterminate value here. */
21      /*
22       * The above return statement exhibits undefined behavior,
23       * because of the access to count.
24       */
25      do_work:;
26      count = glob % 4;
27
28      goto start_block;
29      }
30  }
```

VLA lifetime starts/ends

For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[27)] 464

Commentary

The lifetime of a VLA starts at its point of declaration, not at the point of entry into the block containing its definition. This means it is possible to control the number of elements in the VLA through side effects within the block containing its definition. The intent of the Committee was for it to be possible to implement VLAs using the same stack used to allocate storage for local objects. Although their size varies, the requirements are such that it is possible to allocate storage for VLAs in a stack-like fashion. C does not define any out-of-storage signals and is silent on the behavior if the implementation cannot satisfy the requested amount of storage.

C90

Support for variable length arrays is new in C99.

C++

C++ does not support variable length arrays in the C99 sense; however, it does support containers:

23.1p1 *Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.*

The techniques involved, templates, are completely different from any available in C and are not discussed further here.

Other Languages

The lifetime of arrays in some languages (e.g., Java, Awk, Perl, and Snobol 4) does not start until the declaration, or statement, that contains them is executed. An array definition in Java does not allocate storage for the array elements; it only allocates storage for a reference to the object. The storage for the array elements is created by the execution of an *array access expression*. This can occur at any point where the identifier denoting the declared array is visible. Fortran and Pascal allow variable-size arrays to be passed as arguments. However, this is a subset of the functionality involved in allowing the number of elements in an array definition to be decided during program execution. In PL/1 the storage for arrays whose size is computed during program execution is not allocated at the point of definition, but within the executable statements via the use of the **allocate** statement.

```
1  declare
2       (k, l) fixed bin,
3       a dim (k) char (l) controlled;
4
5  k = 10;
```

```
6   l = 10;
7   allocate a;
8   free a;
```

Common Implementations

Because the size of an object having VLA type is not known at translation time, the storage has to be allocated at execution time. A commonly used technique involves something called a *descriptor*. Storage for this descriptor, one for each object having VLA type, is allocated in the same way as locally defined objects having other types, during translation. The descriptor contains one or more members that will hold information on the actual storage allocated and for multidimensional arrays the number of elements in each dimension. When the definition of an object having a VLA type is encountered during program execution, storage is allocated for it (often on the stack), and the descriptor members are filled in.

```
1   extern void G(void);
2
3   void f(int n)
4   {
5   long l;
6   char a[n];
7   float f_l;
8
9   G();
10  double d[11][n+8];
11  int i_l;
12  /* ... */
13  }
```

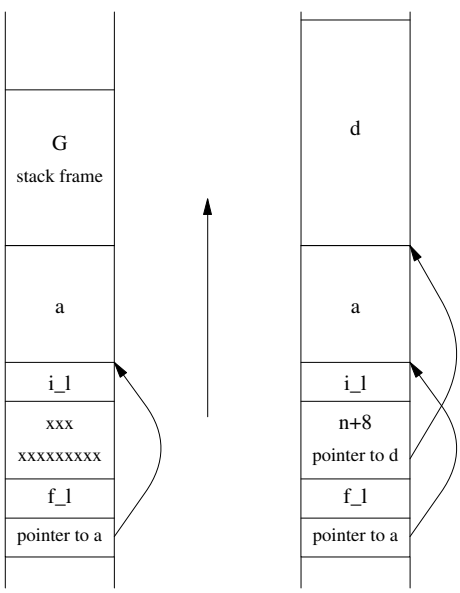


Figure 464.1: Storage for objects not having VLA type is allocated on block entry, plus storage for a descriptor for each object having VLA type. By the time `G` has been called, the declaration for `a` has been reached and storage allocated for it. After `G` returns, the declaration for `d` is reached and is storage allocated for it. The descriptor for `d` needs to include a count of the number of elements in one of the array dimensions. This value is needed for index calculations and is not known at translation time. No such index calculations need to be made for `a`.

Once execution leaves the scope of the object definition, having a VLA type, its lifetime terminates and the allocated storage can be freed up. This involves the implementation keeping track of all constructs that can cause this to occur (e.g., **goto** statements, as well as normal block termination).

One technique for simplifying the deallocation of VLA stack storage is to save the current value of the stack pointer on entry into a block containing VLA object definitions and to restore this value when the block terminates. There are situations where this technique might not be considered applicable (e.g., when a **goto** statement jumps to before a VLA object definition in the same block, an implementation either has to perform deallocation just prior to the **goto** statement or accept additional stack growth).

Coding Guidelines

object 458
lifetime from entry
to exit of block

For objects that don't have a VLA type, jumping over their definition may omit any explicit initialization, but storage for that object will still have been allocated. If the object has a VLA type the storage is only allocated if the definition is executed.

It is not expected that the use of VLA types will cause a change in usage patterns of the **goto** statement. A guideline recommendation dealing with this situation is not considered to be worthwhile.

Example

```
1  extern int n;
2  extern char gc;
3  extern char *pc = &gc;
4
5  void f(void)
6  {
7  block_start:;
8  /*
9   * At this point the lifetime of ca has not yet started.  Jumping back to
10  * this point, from a point where the lifetime has started, will terminate
11  * the lifetime.
12  */
13 n++;
14
15 /*
16  * First time around the loop pc points at storage allocated for gc.
17  * On second and subsequent iterations pc will have been pointing at an
18  * object whose lifetime has terminated (giving it an indeterminate value).
19  */
20 pc[0]='z';
21
22 char ca[n]; /* Lifetime of ca starts here. */
23 pc=ca;
24
25 if (n < 11)
26    goto block_start;
27 }
```

---

object
new instance
for recursion

If the scope is entered recursively, a new instance of the object is created each time. 465

Commentary

block 460
entered recursively

A scope can only be entered recursively through a recursive function call. This behavior is the same as for objects that don't have a VLA type. Functions containing VLAs are reentrant, just like functions containing any other object type.

The functions in the standard library are not guaranteed to be reentrant.

Other Languages

This behavior is common to all block scoped languages.

Common Implementations

In practice few function definitions are used in a way that requires their implementation to be reentrant. Some implementations[1340] assume that functions need not be reentrant unless explicitly specified as such (e.g., by using a keyword such as **`reentrant`**).

---

466 The initial value of the object is indeterminate.

object initial VLA value indeterminate

Commentary

Objects having a VLA type are no different from objects having any other type.

461 object initial value indeterminate

Common Implementations

The usage patterns of objects having a VLA type are not known. Whether they are less likely to have their values from a previous lifetime in a later lifetime than objects having other types is not known.

---

467 **Forward references:** statements (6.8), function calls (6.5.2.2), declarators (6.7.5), array declarators (6.7.5.2), initialization (6.7.8).

---

468 25) The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal.

footnote 25

Commentary

The standard does not specify how addresses are to be represented in a program; it only specifies the results of operations on them. In between their construction and being compared, it is even possible that they are written out and read back in again.

C++

The C++ Standard is silent on this issue.

Other Languages

This statement is true of nearly all classes of languages, although some don't support the construction of pointers to objects. Functional languages (often used when formally proving properties of programs) never permit two pointers to refer to the same object. Assignment of pointers always involves making a copy of the pointed-to object (and returning a pointer to it).

Common Implementations

In some implementations the address of an object is its actual address in the processes (executing the program) address space. With memory-mapping hardware (now becoming common in high-end freestanding environments) it is unlikely to be the same as its physical address. Its logical address may remain constant, but its physical address could well change during the execution of the program.

---

469 The address may be different during two different executions of the same program.

Commentary

It is possible to write a pointer out to a file using the `%p` conversion specifier during one execution of the program and read it back in during a subsequent execution of the program. While the address read back, during the same execution of the program, will refer to the same object (assuming the object lifetime has not ended); however, during a different execution of the program the address is not guaranteed to refer be the same object (or storage allocated for any object).

Addresses might be said to having two kinds of representation details associated with them. There is the bit pattern of the value representation and there is the relative location of one address in relation to another address. Some applications make use of the relationship between addresses for their own internal algorithms. For instance, some garbage collectors for Lisp interpreters depend on low addresses being used for allocated storage. Low address are required because the garbage collector use higher order bits within the address value to indicate certain properties (assuming they can be zeroed out when used in a pointer dereference).

1354 object reserve storage

C++

The C++ Standard does not go into this level of detail.

Other Languages

Most high-level languages do not make visible, to the developer, the level of implementation detail specified in the C Standard.

Common Implementations

Programs running in hosted environments that use a memory-management unit to map logical to physical addresses are likely to use the same logical addresses, to hold the same objects, every time they are executed. It is only when other programs occupy some of the storage visible to a program that the address of its objects is likely to vary.

---

footnote 26 — 26) In the case of a volatile object, the last store need not be explicit in the program. 470

Commentary

volatile 1482 last-stored value — The fact that a volatile object may be mapped to an I/O port, not a storage location, does not alter its lifetime. As far as a program is concerned, the lifetime of an object having a given storage-class is defined by the C Standard.

C++

7.1.5.1p8 *[Note: **volatile** is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of **volatile** are intended to be the same in C++ as they are in C.]*

---

footnote 27 — 27) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration. 471

Commentary

The scope of the declaration is the region of program text in which the identifier is visible using a top-down, left-to-right parse of the source code.

C90

Support for VLAs is new in C99.

Common Implementations

An implementation is required to track all these possibilities. It can choose not to free-up allocated storage in some cases, perhaps because it can deduce that the same amount of storage will be allocated when the definition is next executed. There is little practical experience with the implementation and VLA types at the moment. The common cases can be guessed at, but are not known with certainty.

6.2.5 Types

---

types — The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. 472

Commentary

Without a type, the object representation is simply a pattern of bits. A type creates a value representation from an object representation.

Other Languages

There is a standard for data types: *ISO/IEC 11404:1996 Information Technology— Programming languages, their environments and system software interfaces – Language-independent datatypes*. Quoting from the scope of this standard "This International Standard specifies the nomenclature and shared semantics for a collection of datatypes commonly occurring in programming languages and software interfaces, . . . ".

Some languages, for instance Visual Basic, tag objects with information about their type. When accessed, the implementation uses the tag associated with an object to determine its type, removing the need for the developer to explicitly specify a type for an object before program execution. Some languages (e.g., Algol 68 and CHILL) use the term *mode* rather than *type*.

Coding Guidelines

Accessing the same object using more than one type is making use of representation information and is discussed elsewhere.

569.1 representation information using
948 effective type

---

473 (An identifier declared to be an object is the simplest such expression;

Commentary

Such an expression is a *primary-expression*. Both the form and value of an integer constant determines its type, so an integer constant is not simpler than an identifier.

975 primary-expression syntax
835 integer constant type first in list

---

474 the type is specified in the declaration of the identifier.)

Commentary

C99 does not support the implicit declaration of any identifier. Labels may be used before they are defined, but they must still be defined.

1379 declaration at least one type specifier

Common Implementations

The behavior of some implementations on encountering an identifier that has not been declared is to provide a default declaration (as well as issuing the required diagnostic). Declaring the identifier to be an object of type **int** often prevents cascading diagnostics from being generated. A more sophisticated error-recovery strategy is to examine the token immediately to the left of identifier (adding the identifier as a member of the structure or union type if it is a selection operator).

---

475 Types are partitioned into *object types* (types that fully describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

types partitioned
object types
incomplete types

Commentary

This defines the terms *object types*, *function types*, and *incomplete types*. Function types can never be transformed into another type, although a pointer-to function type is an object type. When an incomplete type is completed, it becomes an object type. In the case of array types it is possible for its complete/incomplete status to alternate. For instance:

```
1  extern int ar[];   /* ar has an incomplete type here.  */
2
3  void f_1(void)
4  {
5  extern int ar[10]; /* ar has a complete type here.    */
6  }
7                     /* ar has an incomplete type here. */
```

Structure and union types can only be completed in the same scope as the original, incomplete declaration.

C++

3.9p6 *Incompletely-defined object types and the void types are incomplete types (3.9.1).*

So in C++ the set of incomplete types and the set of object types overlap each other.

3.9p9 *An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a* ***void*** *type.*

A sentence in the C++ Standard may be word-for-word identical to one appearing in the C Standard and yet its meaning could be different if the term *object type* is used. The aim of these C++ subclauses is to point out such sentences where they occur.

Other Languages

Many languages only have object types. Some languages support pointers to function types, but don't necessarily refer to a function definition as a function type. Incomplete types, in C, were originally needed to overcome the problems associated with defining mutually recursive structure types. Different languages handle this issue in different ways. The Java class method can be viewed as a kind of function type.

_Bool large enough to store 0 and 1

An object declared as type **_Bool** is large enough to store the values 0 and 1. 476

Commentary

Many existing programs contain a type defined, by developers, with these properties. The C committee did not want to break existing code by introducing a new keyword for an identifier that was likely to be in widespread use. **_Bool** was chosen. The header **<stdbool.h>** was also created and defined to contain more memorable identifiers. Like other scalar types, **_Bool** is specified in terms to the values it can hold, not the number of bits in its representation.

While an object of type **_Bool** can only hold the values 0 and 1, its behavior is not the same as that of an **unsigned int** bit-field of width one. In the case of **_Bool** the value being assigned to an object having this type is first compared against zero. In the case of a bit-field the value is cast to an unsigned type of the appropriate width. An example of the difference in behavior is that even, positive values will always cause 1 to be assigned to an object of type **_Bool**, while the same values will cause a 0 to be assigned to the object having the bit-field type. The standard does not prohibit an object of type **_Bool** from being larger than necessary to store the values 0 and 1. The only way of storing a value other than 0 or 1 into such a *larger object* depends on undefined behavior (i.e., two members of a union type, or casting pointer types). Whether, once stored, such a value may be read from the object using its **_Bool** type will also depend on the implementation (which may simply load a byte from storage, or may extract the value of a single bit from storage).

The term *boolean* is named after George Boole whose book, "An investigation of the laws of thought",[137] introduced the concept of Boolean algebra (or *logic* as it is commonly known, a term that is something of an overgeneralization).

C90

Support for the type **_Bool** is new in C99.

C++

3.9.1p6 *Values of type* ***bool*** *are either* ***true*** *or* ***false****.*

An rvalue of type **bool** *can be converted to an rvalue of type* **int**, *with* **false** *becoming zero and* **true** *becoming one.*

A **bool** *value can successfully be stored in a bit-field of any nonzero size.* 9.6p3

Other Languages

Many languages, including Java, support a boolean type; however, they don't usually get involved in specifying representation details. Fortran calls its equivalent type **LOGICAL**.

Common Implementations

Only a few processors contain instructions for loading and storing individual bits— from/to storage. Most implementations have to generate multiple instructions to achieve the required effect. Representing the type **_Bool** using a byte of storage simplifies (shorter, faster code) the loading and storing of its values. Many implementations make this choice.

The Intel 8051[625] has a 16-byte internal data-storage area and supports bit-level accesses to it. Several implementations (e.g., Keil,[717] Tasking[20]) include support for the type specifier **bit**, which enables objects to be declared that denote specific bits in this data area.

Coding Guidelines

George Boole intended his work to " . . . investigate the fundamental laws of those operations of the mind by which reasoning is performed.". It was some time before people realized that the reasoning carried out by the human mind is based on principles other than mathematical logic.

Experience with character types has shown that developers sometimes overlook the effects of promotion, of operands, to the type **int**. It remains to be seen whether objects of type **_Bool** will be used in contexts where such oversights, if made, will be significant. 653 operand convert automatically

Existing programs that define their own boolean type often use the type **unsigned char** as the underlying representation type. Given that implementations are likely to use this representation, internally, for the type **_Bool**, there would not appear to be any worthwhile benefits to changing the underlying type of the developer-defined boolean type. However, translators and static analysis tools are becoming more sophisticated and use of the type **_Bool** provides them with more tightly specified information on the range of possible intended values.

The type **_Bool** is based on concepts derived from boolean algebra and mathematical logic. Other than defining this type, the concept is not discussed again in the standard. Some languages (including C++) specify the result of some operators as having a boolean type. This is not the case in C99 (largely for backwards compatibility with C90). There are 10 kinds of developers, those that understand boolean roles and those that don't.

While the C Standard does not use **_Bool** in the specification of any other constructs, the concept of a boolean role is invariably part of a developer's comprehension of source code. Other terms used by developers when discussing boolean roles include *flag*, *indicator*, *switch*, *toggle*, and *bit*. These terms are usually associated with situations having two different states, or values. True and false, or 0 and 1, are simply different ways of representing these states. Mathematically the representation values could just as well be 222 and 4,567, or 0 and any number except 0. However, in practice the representation does play a part and the minimalist approach is often used (the values 0 and 1 are also representable a bit). 1112 logical negation result type 51 bit

Should the concept of boolean be interpreted in its broadest sense, with any operation that can only ever deliver one of two results be considered as boolean? Or should the C definition of the type **_Bool** be used as the sole basis for how the concept of boolean is interpreted? The former interpretation probably corresponds more closely with how developers think about boolean concepts, while the latter avoids subtle problems with different representations of the two values used in the representation. However, there is no evidence to suggest that either interpretation is unconditionally better than the other.

At the source code level the difference in specification of behavior for boolean and the other arithmetic types can be a small one. In all but the first case C defines the result of the following operations have an integer type (which differs from C++ where some of them have type **bool**):

- Cast of an expression to type **_Bool**.
- The result of a relational, equality, or logical operator.
- The definition of an enumerated type containing two enumeration constants.
- The definition of a bit-field of width one.
- The result of the remainder operator when the denominator has a value of 2.

Is there a worthwhile benefit in a guideline recommendation that classifies certain kinds of operations as delivering a result that has a boolean role and places restrictions on the subsequent use of these results?

The type **_Bool** is new in C99, and there is little experience available on the kinds of developer comprehension costs associated with in its use. However, a boolean type has long been supported in other languages. Is there anything to be learned from this other language usage? The other languages containing a boolean type, known to your author, do not promote operands (having this type) to an integer type. Because the operands retain their boolean type, some of the operations (e.g., addition) available to the C developer are not permitted (without explicitly casting to an integer type). Creating a stricter type system for C which treated the type **_Bool** as being distinct from the integer types and specified a result type of **_Bool** for some operators (e.g., logical negation, relational, equality, logical-AND, and logical-OR), is likely to be a nontrivial exercise that will touch nearly all aspects of the language specification. The approach taken in these coding guidelines is to introduce the concept of roles (leaving the C type system alone) and provide guideline recommendations on the contexts in which objects having these roles may occur.

typedef 1633 is synonym
logical negation 1111 result is
relational operators 1211 result type
equality operators 1220 result type
&& 1252 result type
|| 1260 result type
object 1352 role
boolean role
identifier 792 selecting spelling

An object having an integer type integer type has a boolean role if it is only expected to hold one of two possible values or it is assigned objects that have a boolean role.

The semantic associations implied by boolean usage could suggest spellings of identifiers denoting states that only ever take on one of two values. This issue is discussed elsewhere.

Example

```
#include <stdio.h>

extern _Bool E;

void f(void)
{
if (((E ? 1 : 0) != (!(!E))) ||
    ((E += 2) != 1)          ||
    ((--E, --E, E) != E))
   printf("This is not a conforming implementation\n");
}

void g(void)
{
_Bool Q;
struct {
        unsigned int u_bit :1;
       } u;
#define U (u.ubit)

Q = 0;     U = 0;   // sets both to 0
Q = 1;     U = 1;   // sets both to 1
Q = 4;     U = 4;   // sets Q to 1, U to 0
```

```
24  Q = 0.5;   U = 0.5; // sets Q to 1, U to 0
25  Q++;       U++;     // sets Q to 1; sets U to 1-U
26  Q--;       U--;     // sets Q to 1-Q; sets U to 1-U
27  }
```

---

477 An object declared as type **char** is large enough to store any member of the basic execution character set.

char hold any member of execution character set

Commentary

This requirement on the implementation is not the same as that for byte. One is based on storage and the other on type. The values of the CHAR_MIN and CHAR_MAX macros delimit the range of integer values that can be stored into an object of type **char**.

53 byte addressable unit
311 CHAR_MIN
312 CHAR_MAX

C++

> *Objects declared as characters (**char**) shall be large enough to store any member of the implementation's basic character set.* 3.9.1p1

The C++ Standard does not use the term character in the same way as C.

59 character single-byte

Other Languages

Many languages have some form of type **char**. Objects declared to have this type are capable of representing the value of a single character.

Coding Guidelines

The terms *char* (a type) and *character* (a bit representation) are often interchanged by developers. While technically they have different meanings in C (but not C++), there does not appear to be anything to be gained by educating developers about the correct C usage.

59 character single-byte

---

478 If a member of the basic execution character set is stored in a **char** its value is guaranteed to be ~~positive~~non-negative.

basic character set positive if stored in char object

Commentary

This guarantee only applies during program execution (i.e., it does not apply during preprocessing). The nonnegative value is also representable in a byte. No standardized character set specifies a negative value for its member representations. However, this is not a requirement on the character set an implementation may use. It is a requirement on an implementation that its representation of the type **char** be capable of supporting its chosen basic execution character set.

1883 basic character set may be negative
222 basic character set fit in a byte

The wording was changed by the response to DR #216.

C++

The following is really a tautology since a single character literal is defined to have the type **char** in C++.

> *If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character.* 3.9.1p1

The C++ does place a requirement on the basic execution character set used.

> *For each basic execution character set, the values of the members shall be non-negative and distinct from one another.* 2.2p3

Other Languages

Few languages say anything about the values of character set members. But they invariably use the same character sets as C, so the above statement is also likely to be true for them.

Common Implementations

Although the basic execution character set has less than 127 members, it is possible for a character set to use values outside of the range 0 to 127 to represent members (and EBCDIC does).

Coding Guidelines

Dev 569.1 A program may rely on the value of a member of the basic execution character set stored in an object of type **char** being positive.

If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type. 479

Commentary

This is a requirement on the implementation. They cannot, for instance, specify the implementation-defined value of `'@'` as 999 when objects of type **char** can only represent values between -128 and 127 (but such an implementation-defined value would be possible if the type **char** supported a range of values that included 999).

An implementation supporting a character set containing a member whose value was greater than 127, assuming an 8-bit **char**, would have to chose a representation for the type **char** that had the same range as type **unsigned char**. Such character sets occur within mainstream European languages. The numeric values assigned, by ISO 10646, to characters in the ISO 8859–1 (Latin-1) standard fall in the range 32 to 255.

ISO 8859 24

C90

If other quantities are stored in a ***char*** *object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.*

The implementation-defined behavior referred to in C90 was whether the values are treated as signed or nonnegative, not the behavior of the store. This C90 wording was needed as part of the chain of deduction that the plain char type behaved like either the signed or unsigned character types. This requirement was made explicit in C99. In some cases the C90 behavior for storing *other characters* in a **char** object could have been undefined (implicitly). The effect of the change to the C99 behavior is at most to turn undefined behavior into implementation-defined behavior. As such, it does not affect conforming programs.

char 516 range, representation and behavior

Issues relating to this sentence were addressed in the response to DR #040, question 7.

C++

2.2p3 *The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.*

glyph 58 The only way of storing a particular character (using a glyph typed into the source code) into a char object is through a character constant, or a string literal.

2.13.2p1 *An ordinary character literal that contains a single* `c-char` *has type* ***char****, with value equal to the numerical value of the encoding of the* `c-char` *in the execution character set.*

2.13.4p1

An ordinary string literal has type "array of n `const char`*" and static storage duration (3.7), where n is the size of the string as defined below, and is initialized with the given characters.*

If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. 3.9.1p1

Taken together these requirements are equivalent to those given in the C Standard.

Other Languages

Few language specifications get involved in the representational details of character set members. In most strongly typed languages character literals have type **char**, so by definition they can be represented in a **char** object.

Coding Guidelines

569.1 representation information using

The guideline recommendation dealing with the use of representation information is applicable here. An example of a situation where implicit assumptions on representation may be made is array indexing. A positive value is always required, and the guarantee on positive values only applies to members of the basic execution character set.

Example

```
1   char ch_1 = '@';
```

Usage

In the visible form of the `.c` files 2.1% (`.h` 2.9%) of characters in character constants are not in the basic execution character set (assuming the Ascii character set representation is used for escape sequences).

---

480 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**.

standard signed integer types

Commentary

This defines the term *standard signed integer types*. Having five different types does not mean that there are five different representations (in terms of number of bits used). Multiple integer types, based on processor implementation characteristics, are part of the fabric of C. An implementation could choose to implement all types in 64 bits. They would still be different types, irrespective of the underlying representation. On processors that do not support 32- or 64-bit (needed for **long** and **long long** respectively) integer operations in hardware, there is a strong incentive not to use types requiring this number of representation bits.

There are three real floating types.

497 floating types three real

C90

Support for the type **long long int** (and its unsigned partner) is new in C99.

C++

Support for the type **long long int** (and its unsigned partner) is new in C99 and is not available in C++. (It was discussed, but not adopted by the C++ Committee.) Many hosted implementations support these types.

Other Languages

Many languages only support a single integer type. Some implementations of these languages, for instance Fortran, include extensions that allow the size of the integer type to be specified. Fortran 90 contains the intrinsic function `SELECT_INT_KIND` which enables the developer to specify the range of powers of 10 that an integer should be able to represent. The intrinsic `SELECT_REAL_KIND` allows the decimal precision and exponent range to be specified.

The Ada Standard gives permission for an implementation to support the optional predefined types **SHORT_INTEGER** and **LONG_INTEGER**. Java contains the types **short**, **int**, and **long**. It also specifies that **long** is represented in 64 bits, so there is no need (at least yet) to specify a **long long** that may be larger than a **long**.

Common Implementations

There are commonly at least four bit widths used to represent these types— 8, 16, 32, and 64. On processors that do not support a 64-bit data type (needed for the type **long long**) an implementation will perform 64-bit operations via calls to internal library functions. Some processors do not even support 32-bit operations. Like the 64-bit case, the support is provided via calls to internal library functions.

Coding Guidelines

The fact that there are at least five (four in C90) integer types in C, and not one, is a cause of great misunderstanding, by developers and unintentional behavior, in programs. The existence of these different types causes implicit conversions to become an important issue.

operand 653 convert automatically

There is a simple solution: Restrict programs to using a single integer type. The argument against this solution is often based on a perceived lack of resources (additional time to execute a program and greater storage requirements). The quest for efficiency is often uppermost in developers' minds when deciding on which integer type to use. For instance, an object taking on a small range of values, less than 128 say, is almost automatically given a character type. On RISC- and Intel Pentium-based processors the time taken to load or store an 8- or 32-bit quantity is usually the same. Once the value is loaded into a register, it is often operated on as a 32-bit quantity. There are 8-bit operations available on the Pentium, but they are not often generated by translators because of the extra complication needed for what can be small savings.

On the low-cost processors found in many freestanding environments, there are often resource limitations. Use of a narrower integer type can result in worthwhile performance improvements and storage savings. Implementations for such environments usually use the same representation for the types **int** and **short**. In a hosted environment the representation for the type **int** is usually the same as for the type **long**. The choice of representation for the type **int** is intended to be natural to the execution environment.

int 485 natural size

Rev 480.1 The type specifiers **char**, **short**, and **long** shall not appear in the visible source, or be used via developer-defined macros or via developer-defined typedef names.

Dev 480.1 When resources are constrained and where a worthwhile benefit has been calculated a character type whose promoted type is **int** may be used.

Dev 480.1 If a program needs to represent an integer value that is outside the representable range of the type **int** the type **long** or **long long** type may be used.

Dev 480.1 If a program needs to represent an integer value that is outside the representable range of the type **int**, resources are constrained and where a worthwhile benefit has been calculated, the type **unsigned int** may be used.

Developers rarely have any control over the contents of system headers. These may contain typedefs, for instance `size_t`, that have an integer type other than **int**. While use of these system-defined typedefs has many advantages, they do provide an alternative route for operands having different integer types to appear within expressions.

It is possible for values of different types to occur in an expression, other than via accesses to declared objects. For instance, an operand might be a wide character constant, the result of the **sizeof** operator, or the result of subtracting two pointers.

wide character constant 887 type of sizeof 1127 result type pointer 1175 subtract result type integer types 303 sizes

The minimum range of values that can be represented in these integer types is specified by the standard. An algorithm may depend on some object being able to represent a particular range of values. Use of a typedef name, in declarations of objects, provides a single point of control for the underlying representation

(the typedef name definition). The issue of encoding this range information in the typedef name is discussed elsewhere.

792 typedef naming conventions

Some coding guideline documents recommend that these keywords not be used directly in declarations (e.g., MISRA rule 13). Instead, it is recommended that typedef names denoting integer types of specific widths be defined and used. Such typedefs can be applicable in some situations. However, the number of programs where it is necessary to be concerned about the widths of all integer object representations is small. A much more important association is implied by the semantics associated with typedef names, this is discussed elsewhere.

0 MISRA

673 typedef assumption of no integer promotions

Usage

It is possible to specify many of the integer types, in C, using more than one sequence of keywords. Usage information on integer types is given elsewhere (see Table 1378.1).

1629 typedef name syntax

---

481 (These and other types may be designated in several additional ways, as described in 6.7.2.)

Commentary

Useful for developers who are lazy typists, entrants to the Obfuscated C contest,[43] and writers of coding guideline recommendations who want to pad out their material.

1378 type specifier syntax

Other Languages

Most languages define a single way of specifying every type.

---

482 There may also be implementation-defined *extended signed integer types*.[28)]

extended signed integer types

Commentary

The Committee recognized that new processors are constantly being developed to fill niche markets. These markets do not always require the characteristics normally expected of a general-purpose processor. Sometimes special-purpose uses result in unusual architectural designs. Rather than have vendors extending the language in different kinds of ways, the Committee has attempted to provide a standard framework for extended integer types. The **<stdint.h>** header is one place where these extended integer types may be defined.

Is using " . . . implementation-defined extended signed integer types." making use of an extension or is it implementation-defined behavior? It is listed as an implementation-defined behavior in annex J.3.5.

An implementation may also define new keywords to denote the existing types specified in the standard.

512 footnote 34

C90

The C90 Standard never explicitly stated this, but it did allow extensions. However, the response to DR #067 specified that the types of `size_t` and `ptrdiff_t` must be selected from the list of integer types specified in the standard. An extended type cannot be used.

C++

The C++ Standard does not explicitly specify an implementation's ability to add extended signed integer types, but it does explicitly allow extensions (1.4p8).

Other Languages

The Pascal/Ada language family allows developer-specified subranges (lower and upper bounds on the possible values) of integer types to be defined. These subranges provide information to the translator that enable it to select the appropriate underlying, processor, integer type to use.

Coding Guidelines

If the guideline recommendation dealing with using a single integer type is followed, any extended types will not occur in the source.

480.1 object int type only

Having occurrences of a nonstandard type, supported by a particular implementation, scattered throughout the visible source code creates portability problems. A method of minimizing the number of visible occurrences is needed. Use of a typedef name does not necessarily offer the best solution:

```
/*
 * Assuming vendor_int is a keyword denoting an extended
 * integer type supported by some implementation.
 */
#if USING_VENDOR_X
typedef vendor_int INT_V;
#else
typedef int INT_V;
#endif

unsigned INT_V glob; /* Syntax violation. */
```

type specifier syntax 1378

C syntax does not allow a declaration to include both a typedef name and another type specifier. Using a macro name to cover the implementation-defined keyword would avoid this problem.

---

signed integer types

The standard and extended signed integer types are collectively called *signed integer types*.[29)] 483

Commentary

This defines the term *signed integer types*. Unlike the integer types, other types do not have unsigned versions (floating-point types are always signed) and are not thought of in terms of being signed or unsigned. Hence the term commonly used is *signed types* rather than *signed integer types*.

C90

Explicitly including the extended signed integer types in this definition is new in C99.

An object declared as type **signed char** occupies the same amount of storage as a "plain" **char** object. 484

Commentary

character types 515

The amount of storage occupied by the two types may be the same, but they are different types and may be capable of representing different ranges of values. The representation of the sign in the type **signed char** is part of the object representation and is not permitted to consume additional bits.

---

int natural size

A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header **<limits.h>**). 485

Commentary

A "plain" **int** is the most commonly used type in source code. The choice of its representation has many important consequences, particularly in the quality of machine code produced by a translator. There may be several interpretations of the term *natural size* for some processor architectures. Efficiency is usually a big consideration. Execution-time performance efficiency is not always the same as storage efficiency for a given architecture. An implementors choice of "plain" **int** also needs to consider its effect on how integer types with lower rank will be promoted.

ABI

The choice of representation need not be decided purely on how the available processor instructions manipulate their operands. In function calls the most commonly passed argument type is "plain" **int**. The organization of the function call stack is an important design issue. In some cases it may have already been specified by a hardware vendor.[613, 1345–1347, 1517] In this case the choice of "plain" **int** representation has already been dictated for all translators targeting that environment.

Other Languages

Most languages say nothing about the representation of integer types; it is left to implementation vendors to decide how best to implement them for a given host. Java is designed to be portable across all environments. This is achieved by specifying the representation of all types in the language standard.

Common Implementations

Historically the "plain" **int** type was usually the same size as either the types **short** or **long**. It is rare to find an **int** type having a size that is between the sizes of these two types. This existing practice has affected

existing code, which is often found to contain implicit assumptions about various integer types having the same widths. 626 width integer type

Coding Guidelines

Developers sometimes want the program they write, or at least large parts of them, to be independent of the host processor (the current market dominance of the Intel/Microsoft platform has caused many development groups to lose interest in portability to other platforms). Having a fundamental integer type whose representation depends on natural features of the host environment does not sound ideal. One solution is to hide implementation decisions behind a typedef name and recommend that developers use this rather than the **int** keyword (moving source code to a new implementation then only requires one change to the typedef definition).

It used to be the case that both 16- and 32-bit processors were commonly encountered in a hosted environment. The migration, for hosted environments, to 32-bit processors is now an established fact. The migration to processors where 64-bit integers are the natural architectural choice has only just started (in 2002). Whether it is yet worth investing effort on enforcing a guideline that recommends using a typedef name rather than the **int** keyword is uncertain (for programs aimed at the desktop environment).

Mobile phones and hand-held organizers are starting to support the execution of programs downloaded by their users. To conserve power and reduce costs, some of the processors used in such devices have a natural integer size of 16 bits. The extent to which a large number of programs will need to be ported from a 32-bit environment to these hosts is unknown.

The processors used in freestanding environments vary enormously in their support for different sized integer types. On some, support for a 16-bit **int** (the minimum standard requirement) does not come naturally. At the time of this writing a large amount of the software written for these processors does not get ported to other processors, it is device-specific (as much driven by the characteristics of the application as by the cost of changing processors). But developer organizations do want to reuse code if possible— it can reduce time to market (but not always total cost). In this development environment, where the size of the type **int** is likely to vary, there are obvious benefits in a guideline recommendation to use a typedef name.

The guideline recommendation dealing with using a single integer type specifies the use of the type **int**. 480.1 object int type only

---

486 For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. signed integer corresponding unsigned integer

Commentary

This is another case of C providing a construct that mirrors a data type, and associated operations, commonly found in hardware processors. The unsigned integer types can generally represent positive values twice as great as their corresponding signed counterparts.

Other Languages

Few languages support unsigned integer types. Modula-2 uses the term *cardinal* to denote its unsigned integer type. Support for what were called *modular* types was added to Ada 95.

Coding Guidelines

The mixing of operands having unsigned and signed integer type in an expression is a common cause of unexpected program behavior. Developers overlook the effects of performing the integer promotions clauses and usual arithmetic conversions, or are simply unaware that the operands have different signedness. This issue is also discussed elsewhere. 675 integer promotions 706 usual arithmetic conversions 653 operand convert automatically

Following the guideline recommendation that only the type **int** be used implies that no objects having an unsigned type are declared. Does a deviation from this guideline recommendation have a worthwhile benefit? Measurements show (see Table 486.1) that unsigned types are much more common in embedded software than signed types. The fact that some languages do not have an unsigned type might be more of an indication 480.1 object int type only

of the applications written using them than that programs don't need to use such a type. The following are several ways of thinking about unsigned types in relation to signed types:

integer types 303 sizes

integer promotions 675

- Signed types are the *natural type* to use and any usage of unsigned types needs to be justified. In many programs the range of values manipulated is well within the representable range of the integer types used. The ability of signed types to represent negative values frees developers of the need to think about the possibility of having to handle negative values created as the intermediate step during the evaluation of an expression. The type **`int`** is the type to which narrower types are converted by the integer promotions. The type **`int`** is equivalent to the type **`signed int`**. Signed types appear to be the type of least effort (for developers).
- Unsigned types are the *natural type* to use and any usage of signed types needs to be justified. Most quantities that are counted or measured have positive values (mathematicians refused to acknowledge the existence of negative quantities for many centuries[138]). In many cases applications do not deal with negative quantities. Unsigned types appear to share the same *positive quantity* semantics as applications.
- Neither type is considered more natural to use than the other. There does not appear to be any situation where this would be the default position to take.

Using the type **`unsigned int`** only prevents signed/unsigned conversion issues from arising if it is the only type used. Objects declared using a type of less rank will be promoted to the type **`int`**.

Alt 480.1

All objects declared in a program shall have an unsigned type.

Developers often only consider the use an unsigned type when the possible range of values being stored is not representable in the corresponding signed type. There may not be a signed type capable of representing the desired range of values (support for the type **`long long`** is new in C99), or use of this type may be considered to be *inefficient*. Whether signed/unsigned conversion issues are minimized by declaring all objects to have either a signed or unsigned type will have to be decided by the developer.

Usage

Usage information on integer type specifiers is given elsewhere (see Table 1378.1, which does not include uses of integer types specified via typedef names).

Table 486.1: Occurrence of objects having different width integer types (as a percentage of all integer types) for embedded source and the SPECint95 benchmark (separated by a forward slash, e.g.,embedded/SPECint95). Adapted from Engblom.[390]

	8 bits	16 bits	32 bits
unsigned	70.8/1.3	14.0/0.4	2.1/44.9
signed	2.7/0.0	9.4/0.3	1.0/53.1

A study by Engblom[390] compared the use of integer types in embedded source and SPECint95 (see Table 486.1). There are a number of possible reasons of Engblom's results. Most of the difference is due to the use of the type **`unsigned char`**. There can be significant performance advantages in using this type (because of processor characteristics <host processors, introduction>). There is also the observation that measurements of physical quantities are usually positive quantities. Other reasons include the following:

- Developers writing for the desktop are not motivated to spend time tuning the sizes of scalar objects. (The fact that any storage saving from using a smaller type is insignificant and the instruction timings for the different types is often very similar if not identical is not the issue, because many developers believe there are performance differences.) There are often few resources (e.g., developers time, performance-monitoring tools) provided by management to motivate developers to think about performance issues in this environment.

- Processors used for embedded applications usually have instructions specifically designed to efficiently handle 8-bit data. Programs often have storage constraint when executing in such environments. There are real benefits to be had from being able to use an 8- or 16-bit data type. The technical case, showing that performance improvements are available, is visible to developers and their managers. The demands of the application can also require management to make the resources available to consider performance as the application is being designed, coded, and maintained.

It might be claimed that embedded applications are oriented to 8-bit data. The application requirements control the choice of processor, not the other way around. Hardware cost is important (designers worry over pennies, which add up when millions of units are to be manufactured). Processors that handle 8-bit data are usually cheaper to manufacture and the interfacing costs are lower (fewer wires to the outside world).

---

487 The type **_Bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. standard unsigned integer

Commentary

This defines the term *standard unsigned integer type*.

The unsignedness of the type **_Bool** is unlikely to be visible to the developer. The usual arithmetic conversions will always promote objects having this type to the type **int**. 706 usual arithmetic conversions 476 _Bool large enough to store 0 and 1

C90

Support for the type **_Bool** is new in C99.

C++

In C++ the type **bool** is classified as an integer type (3.9.1p6). However, it is a type distinct from the signed and unsigned integer types.

Coding Guidelines

Although the type **_Bool** may, technically, be an integer type, there are benefits to treating it as a distinct type. However, there is a potential cost in doing so. Treating it as a distinct type reinforces the (technically incorrect) developer expectation that implementations will treat it as purely a boolean type, having one of two values. If a value other than 0 or 1 (through use of a construct exhibiting undefined behavior) is stored in such an object, implicit assumptions in a program may no longer hold. These coding guidelines assume that the benefits of treating the type **_Bool** as a distinct type significantly outweigh the potential costs The fact that the type **_Bool** is classified as an unsigned integer type is not of any practical significance to coding guidelines. 476 boolean role

---

488 The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. extended unsigned integer

Commentary

This defines the term *extended unsigned integer type*. By definition any unsigned integer type that does not have a corresponding extended signed integer type is not as an *extended unsigned integer type*.

Common Implementations

The concept of extended integer types is still too new to be able to say anything about common implementations. The Motorola AltiVec implementation[971] supports the type **pixel**. This uses 16 bits to represent a display pixel (a 1/5/5/5 bit interpretation of the bits is used, and the type is treated like an **unsigned short**).

Coding Guidelines

The use of extended unsigned integer types violates the guideline recommendation dealing with using extensions and using a single integer type. Such types are not usually provided by implementations. Support for them, by an implementation, implies customer demand. Given such demand it is likely that they will be used, in some cases, by developers. Following coding guidelines is unlikely to take precedence in these cases, and nothing more is said about the issue. 95.1 extensions cost/benefit 480.1 object int type only

unsigned integer types

The standard and extended unsigned integer types are collectively called *unsigned integer types*.[30)] 489

Commentary

This defines the term *unsigned integer types*. It is common practice to use the term *unsigned types* to denote these types.

signed integer types 483

C90

Explicitly including the extended signed integer types in the definition is new in C99.

footnote 28

28) Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3. 490

Commentary

An implementation-defined keyword is an extension, however it is spelled. In specifying how new keywords are to be added to the language the Committee is recognizing that this is often done by translator vendors. By listing the identifier spellings that should be used, the Committee is also giving a warning to developers not to use them in their own declarations.

The incentive for vendors to use these spellings is that they can claim their extensions have been added in a standards-conforming way. What do developers gain from it? Existing code is less likely to be broken (because it is not supposed to use these spellings) and the Committee continues to have the option of using nonreserved words in future language enhancements (the Committee never makes extensions).

C90

The C90 Standard did not go into this level of detail on implementation extensions.

C++

The C++ Standard does not say anything about how an implementation might go about adding additional keywords. However, it does list a set of names that is always reserved for the implementation (17.4.3.1.2).

Other Languages

Most languages say nothing about possible extensions to themselves, perhaps in the hope that there will not be any such extensions. Those that do rarely specify a list of reserved identifiers.

Common Implementations

Suggestions about the spelling of additional keywords was not given in C90 and vendors used a variety of identifiers, only a few of them following the usage described in 7.1.3. One of the most commonly seen keyword extensions, **`asm`**, is not in the list of reserved names. The keywords **`near`** and **`far`** (and sometimes **`tiny`** and **`huge`**) were used by translators targeting the Intel x86 processor when many of the applications were predominantly 16 bit. While most applications for this processor are now 32 bit, support for these keywords is kept for backwards compatibility. These keywords have also been adopted by translators targeting other processors, where pointers of various widths need to be supported.

The token sequence **`long long`** was a relatively common extension in C90 implementations. This usage did not introduce a new keyword; it used existing keywords to create a new type.

typing minimization 0

Experience shows that developers prefer keywords that are short and meaningful, and they often complain that having to prefix keywords with underscores is irksome. A solution used by some implementations that have added extra keywords is to define matching macro names (e.g., `gcc` defines the keywords **`__attribute__`** and **`__typeof__`**; It also predefines the macro names `attribute` and `typedef`, which expand to the respective keywords).

The Keil compiler[717] supported the keyword **`bit`**, for some processors, well before the C99 Standard was published.

footnote 29

29) Therefore, any statement in this Standard about signed integer types also applies to the extended signed integer types. 491

Commentary

Because the extended integer types are included in the definition of signed integer types, any statement that refers to the term *signed integer types* also includes any extended integer types.

C++

The C++ Standard does not place any requirement on extended integer types that may be provided by an implementation.

Common Implementations

It remains to be seen whether implementations abide by this non-normative footnote.

Coding Guidelines

Any guideline recommendation that applies to the signed integer types also applies to any extended signed integer types. The additional guidelines that apply to extended signed integer types are those that apply generally to the use of extensions. The additional complexity introduced into the usual arithmetic conversions needs to be considered.

95 implementation extensions
706 usual arithmetic conversions

---

492 30) Therefore, any statement in this Standard about unsigned integer types also applies to the extended unsigned integer types.

footnote 30

Commentary

The discussion on statements that apply to the signed integer types is also applicable here.

491 footnote 29

---

493 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

standard integer types
extended integer types

Commentary

This defines the terms *standard integer types* and *extended integer types*. Note that this definition does not include the type **char** or the enumerated types. These are included in the definition of the term *integer types*.

519 integer types

C++

Types **bool**, **char**, **wchar_t**, *and the signed and unsigned integer types are collectively called integral types.*[43)] *A synonym for integral type is integer type.*

3.9.1p7

43) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to **int**, **unsigned int**, **long**, *or* **unsigned long**, *as specified in 4.5.*

Footnote 43

The issue of enumerations being distinct types, rather than integer types, is discussed elsewhere.

864 enumeration constant type

---

494 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

integer types relative ranges
rank relative ranges

Commentary

The specification of the sizes of integer types provides a minimum range of values that each type must be able to represent. Nothing is said about maximum values. The specification of rank is based on the names of the types, not their representable ranges. This requirement prevents, for instance, an implementation from supporting a greater range of representable values in an object of type **short** than in an object of type **int**. An implementation could choose to support an integer type, with a given conversion rank, capable of representing the same range of values as an integer type with greater rank. But an integer type cannot be

303 integer types sizes
659 conversion rank
659 conversion rank

capable of representing a greater range of values than another integer type (of the same sign) without also having a greater rank.

The concept of rank is new in C99.

C90

There was no requirement in C90 that the values representable in an unsigned integer type be a subrange of the values representable in unsigned integer types of greater rank. For instance, a C90 implementation could make the following choices:

```
1  SHRT_MAX  ==   32767  /* 15 bits */
2  USHRT_MAX ==  262143  /* 18 bits */
3  INT_MAX   ==   65535  /* 16 bits */
4  UINT_MAX  ==  131071  /* 17 bits */
```

No C90 implementation known to your author fails to meet the stricter C99 requirement.

C++

3.9.1p2 *In this list, each type provides at least as much storage as those preceding it in the list.*

C++ appears to have a lower-level view than C on integer types, defining them in terms of storage allocated, rather than the values they can represent. Some deduction is needed to show that the C requirement on values also holds true for C++:

3.9.1p3 *For each of the signed integer types, there exists a corresponding (but different) unsigned integer type: "**unsigned char**", "**unsigned short int**", "**unsigned int**", and "**unsigned long int**," each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[40] ;*

They occupy the same storage,

3.9.1p3 *that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.*

they have a common set of values, and

3.9.1p4 *Unsigned integers, declared **unsigned**, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.[41]*

more values can be represented as the amount of storage increases.

QED.

Coding Guidelines

Each integer type's ability to represent values, relative to other integer types, is something developers learn early and have little trouble with thereafter. It is developers' expectation of specific integer types being able to represent the same range of values as other integer types that is often the root cause of faults and portability problems.

In a 16-bit environment there is often an expectation that the type **int** is represented in 16 bits, a width that differs from the type **long** and (sometimes) pointer types. In a 32-bit environment there is often an expectation that the pointer types and the types **int** and **long** are represented using the same amount of storage, each being capable of storing any value that can be represented in the other's type.

The introduction of the type **long long** in C99 uncovered a new expectation— that the type **long** is the widest integer type (and the less reasonable expectation that the type **long** occupies at least the same amount of storage as a pointer). The typedef `intmax_t` was introduced to provide a name for the concept of widest integer type to prevent this issue from causing a problem in the future. However, this typedef name does not solve the problem in existing code.

widest type assumption

---

495 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[31)]

positive signed integer type subrange of equivalent unsigned type

Commentary

This is a requirement on the implementation (even though it does not use the term *shall*). The standard is doing more than enshrining various, low-level representation issues within the language. If the type of operands differ in their sign only, the result of the usual arithmetic conversions is an unsigned type. This requirement ensures that such a conversion does not lead to any surprises if the operand with signed type is positive. This requirement does rule out implementations using the following:

706 usual arithmetic conversions

- More representation bits in the signed type; for instance, the signed type using 24 bits and the unsigned type 16 bits.
- Combinations of integer representations; for instance, two's complement for a particular signed type and BCD for its corresponding unsigned type (although the use of BCD is ruled out by other requirements).

610 sign bit representation

However, it does not rule out the possibility of an unsigned integer type being able to represent significantly more values than its corresponding signed type.

623 object representation same padding signed/unsigned

Other Languages

Most languages do not specify representation details down to this level.

Common Implementations

Some host processors contain instructions for operating on BCD representations (e.g.,the Intel x86). A few older processors aimed at the business language (Cobol) market had no instructions for performing arithmetic on binary integer representations (e.g., ICL System 25). C on such hardware would be problematic.

---

496 A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

unsigned computation modulo reduced

Commentary

The common term for such operations is that they *wrap*.

683 unsigned integer conversion to

This specification describes the behavior of arithmetic operations for the vast majority of existing processors operating on values having unsigned types. Modulo arithmetic is a long-established part of mathematics (number theory). However, mathematicians working in the field of program correctness often frown on reliance on modulo arithmetic operations in programs (they represent a discontinuity). The **%** operator provides another mechanism for obtaining modulo behavior.

1149 % operator result

The LIA (Language Independent Arithmetic) standards[642] treat both nonrepresentable signed and unsigned values in the same way, a mathematical view of the world where values are never intended to exceed their maximum values. C's view is based on how existing processors operate on unsigned quantities.

Other Languages

Languages that support an unsigned integer type usually specify a similar behavior.

Common Implementations

This behavior describes what most processor operations on unsigned values already do. In those rare cases where a processor does not support unsigned operations, calls to internal library functions have to be made.

Coding Guidelines

Are the people working in the field of proving program correctness themselves correct to recommend against the use of modulo arithmetic? It seems to your author that such people are more interested in making programs fit their favorite mathematical theories and algorithms than making the mathematics fit the programs that commercial developers write. In practice the truncation of significant bits (which is how the engineers who originally designed the processor operations thought about it) for operations on unsigned values does not differ from the behavior commonly seen for signed values. However, the standard specifies a well-defined behavior in the former case and undefined behavior in the latter case. However, the difference is not in the commonly seen behaviors, or the specification provided in the standard. The difference is in the commonly seen developer intent.

arithmetic 607 operation
exceptional condition
exception 947 condition

Developers sometimes write code that relies on the modulo behavior of operations on unsigned values. It is very rare for developers to write code that relies on the common processor behavior on signed value overflow. It is not possible to imply from an object having an unsigned type that developers intend operations using it to wrap. The fact that modulo arithmetic is defined by the standard and used by developers some of the time does not mean that this behavior is always intended.

Are there any common cases? Are objects usually defined with unsigned integer types purely to make use of the larger range of values available, or is the modulo behavior on overflow intended? A similar question can be asked of a cast to an unsigned integer type.

It is your author's experience (who has no figures to back his view up) that most operations whose mathematical result is not within the range of the type are not intended by developers. Wanting the result to be modulo-reduced is the less common case. For this reason (and the effects of the mathematical puritans) many coding guideline documents specify that operations on operands having unsigned type should not generate a value that is different from its modulo-reduced value (i.e., operations on unsigned values are not allowed to *overflow*).

Some algorithms depend on modulo arithmetic behavior (e.g., in cryptography). Recommending against all such usage is not constructive. Documenting all such dependencies is a useful aid to readers because they can then assume that in all other cases no such behavior is intended.

Rev 496.1 Those operations involving operands, having an unsigned integer type, where it is known that the result may need to be modulo-reduced shall be commented as such.

---

floating types three real

There are three *real floating types*, designated as **float**, **double**, and **long double**.[32)] 497

Commentary

This defines the term *real floating types*. The possibility of additional floating-point types is listed in future language directions

floating types 2034 future language directions

The organization of the floating types has a similar structure to that commonly seen in the handling of the integer types **short**, **int**, and **long**. The type **double** is often thought of in terms of the floating-point equivalent of **int**. On some implementations it has the same size as the type **float**, on other implementations it has the same size as the type **long double**, and on a few implementations its size lies between these two types. One difference between integer and floating types is that in the latter case an implementation is given much greater freedom in how operations on operands having these types are handled. The header **<math.h>** defines the typedefs `float_t` and `double_t` for developers who want to define objects having types that correspond to how an implementation performs operations.

standard 480 signed integer types
354 FLT_EVAL_METHOD

The type **long double** was introduced in C90. It was not in K&R C.

base document 1

C90

What the C90 Standard calls *floating types* includes complex types in C99. The term *real floating types* is new in C99.

Other Languages

Many languages only support a single floating-point type. Some implementations of these languages, for instance Fortran, include extensions that allow the size of the floating-point type to be specified. The Fortran 90 standard provides a mechanism for developers to specify decimal precision and exponent range. The Ada Standard gives permission for an implementation to support the optional predefined types **SHORT_FLOAT** and **LONG_FLOAT**. Java has the types **float** and **double** type, but does not have a type **long double**.

Common Implementations

The original K&R specification treated the token sequence **long float** as a synonym for **double** (this support was removed during the early evolution of C[1180] although some implementations continue to support it[600]). However, some vendors continue to support this usage.[612]

When floating-point operations are performed using hardware, there are usually two or more different floating-point formats (number of bits). When these operations are carried out in software, sometimes only a single representation is available (in some cases the vendor only provides floating-point support to enable them to claim conformance to the C Standard). The IEC 60559 Standard defines single- and double-precision formats. It also supports an extended precision form for both of these representations.

29 IEC 60559

The Cray processors are unusual in that the types **float** and **double** are both represented in the same number of bits. It is the size of **float** that has been increased to 64 bits, not **double** reduced to 32 bits. The IBM ILE C compiler[617] supports a packed decimal data type. The declaration `decimal(6, 2) x;` declares `x` to have six decimal digits before the decimal point and two after it.

Coding Guidelines

The simple approach of using as much accuracy as possible, declaring all floating-point objects as type **long double**, does not guarantee that algorithms will be well behaved. There is no substitute for careful thought and this is even more important when dealing with floating-point representation.

The type **double** tends to be the floating-point type used by default (rather like the type **int**). Execution time performance is an issue that developers often think about when dealing with floating-point types, sometimes storage capacity (for large arrays) can also be an issue. The type **double** has traditionally been the recommended floating type, for developers to use by default, although in many cases the type **float** provides sufficient accuracy. Given the problems that many developers have in correctly using floating types, a more worthwhile choice of guideline recommendation might be to recommend against their use completely.

It may be possible to trade execution-time performance against accuracy of the final results, but this is not always the case. For instance, some processors perform all floating-point operations to the same accuracy and the operations needed to convert between types (less/more accurate) can decrease execution performance. For processors that operate on formats of different sizes, it is likely that operations on the smaller size will be faster. The question is then whether enough is understood, by the developer, about the algorithm to know if the use a floating-point type with less accuracy will deliver acceptable results.

353 floating-point architectures

In practice few algorithms, let alone applications, require the stored precision available in the types **double** or **long double**. However, a minimum amount of accuracy may be required in the intermediate result of expression evaluation. In some cases the additional range supported by the exponents used in wider types is required by an application.

354 FLT_EVAL_ME
334 exponent

Given the degree of uncertainty about the costs and benefits in using any floating types, this coding guideline subsection does not make any recommendations.

Table 497.1: Occurrence of floating types in various declaration contexts (as a percentage of all floating types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
float	35.2	15.1	8.3	0.7	21.0	80.3
double	8.5	7.9	0.5	0.7	2.2	19.7
long double	0.0	0.0	0.0	0.0	0.0	0.0
Total	43.6	22.9	8.8	1.5	23.2	

The set of values of the type **float** is a subset of the set of values of the type **double**; 498

Commentary

This is a requirement on the implementation. There are no special values that can be represented in the type **float** that cannot also be represented in the type **double**.

Common Implementations

In IEC 60559 the significand, in double-precision, contains more representation bits than in single-precision (assuming these representations are chosen for the types **float** and **double**). Similarly, the exponent can represent a greater range of powers of 2.

Coding Guidelines

With 30 additional bits of significand in IEC 60559, there is a 1 in 10^9 chance of a value represented in double-precision having an exact value presentation in single-precision (leaving aside the possible extra exponent range available in the type **double**).

double values subset of long double

the set of values of the type **double** is a subset of the set of values of the type **long double**. 499

Commentary

This is a requirement on the implementation. There are no special values that can be represented in the type **double** that cannot also be represented in the type **long double**.

Common Implementations

On an Intel x86-based host the type **double** is usually 64 total bits. Some implementations also choose this representation for the type **long double**. Others make use of the extended precision mode supported by the floating-point unit, which is used for performing all floating-point operations. In these cases the type **long double** is often represented by 80 or 96 bits.

For 128-bit **long double** most IEC 60559 implementations use the format of 1–15–113 bits for sign-exponent-significand (and a hidden bit just like single and double).

floating types 330 characteristics

Some implementations use two contiguous **double**s to represent the type **long double**. This format has problems near DBL_MIN in that no extra precision, than available in the type **double**, is available. Also, while it can represent 2*DBL_MAX, there is normally no additional range available. There is also the issue of the interpretation of LDBL_EPSILON.

*_EPSILON 377

complex types

There are three *complex types*, designated as **float _Complex**, **double _Complex**, and **long double _Complex**.[33] 500

Commentary

The introduction of complex types in C99 was based on a marketing decision. The Committee wanted to capture the numerical community, which was continuing to use Fortran. The Committee was told that one of the major reasons engineers and scientists prefer Fortran is that it does a better job of supporting their requirements for numerical computation, support for complex types being one of these requirements. Fortran does not support complex integer types and there did not seem to be sufficient utility to introduce such types in C99.

A new header, **<complex.h>**, has been defined for performing operations on objects having the type **_Complex**. Annex G (informative) specifies IEC 60559-compatible complex arithmetic.

C90

Support for complex types is new in C99.

C++

In C++ complex is a template class. Three specializations are defined, corresponding to each of the floating-point types.

The header **<complex>** *defines a template class, and numerous functions for representing and manipulating complex numbers.*

The effect of instantiating the template `complex` *for any type other than float, double, or long double is unspecified.* 26.2p2

The C++ syntax for declaring objects of type **complex** is different from C.

```
#ifdef __cplusplus

#include <complex>

typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;

#else

#include <complex.h>

typedef float complex float_complex;
typedef double complex double_complex;
typedef long double complex long_double_complex;
#endif
```

Other Languages

Fortran has contained complex types since an early version of that standard. Few other languages specify a built-in complex type (e.g., Ada, Common Lisp, and Scheme).

Common Implementations

Very few processors support instructions that operate on complex types. Implementations invariably break down the operations into their constituent real and imaginary parts, and operate on those separately.

`gcc` supports integer complex types. Any of the integer type specifiers may be used.

Coding Guidelines

The use of built-in language types may offer some advantages over developer-defined representations (optimizers have more information available to them and may be able to generate more efficient code). However, the cost involved in changing existing code to use this type is likely to be larger than the benefits reaped.

---

501 The real floating and complex types are collectively called the *floating types*. floating types

Commentary

This defines the term *floating types*.

C90

What the C90 Standard calls *floating types* includes complex types in C99.

Coding Guidelines

There is plenty of opportunity for confusion over this terminology. Common developer usage did not use to distinguish between complex and real types; it did not need to. Developers who occasionally make use of floating-point types will probably be unaware of the distinction, made by the C Standard between real and complex. The extent to which correct terminology will be used within the community that uses complex types is unknown.

For each floating type there is a *corresponding real type*, which is always a real floating type. 502

Commentary

This defines the term *corresponding real type*. The standard does not permit an implementation to support a complex type that does not have a matching real type. Given that a complex type is composed of two real components, this may seem self-evident. However, this specification prohibits an implementation-supplied complex integer type.

For real floating types, it is the same type. 503

Commentary

It is the same type in that the same type specifier is used in both the real and complex declarations.

For complex types, it is the type given by deleting the keyword **_Complex** from the type name. 504

Commentary

The keyword **_Complex** cannot occur as the only type specifier, because it has no implicit real type. One of the real type specifiers has to be given.

C++

In C++ the complex type is a template class and declarations involving it also include a floating-point type bracketed between < > tokens. This is the type referred to in the C99 wording.

complex type representation

Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; 505

Commentary

This level of specification ensures that C objects, having a complex type, are likely to have the same representation as objects of the same type in Fortran within the same host environment. Such shared representations simplifies the job of providing an interface to library functions written in either language.

Rationale

The underlying implementation of the complex types is Cartesian, rather than polar, for overall efficiency and consistency with other programming languages. The implementation is explicitly stated so that characteristics and behaviors can be defined simply and unambiguously.

C++

The C++ Standard defines `complex` as a template class. There are no requirements on an implementation's representation of the underlying values.

Other Languages

Other languages that contain complex as a predefined type do not usually specify how the components are represented in storage. Fortran specifies that the type **complex** is represented as an ordered pair of real data.

Common Implementations

Some processors have instructions capable of loading and storing multiple registers. Such instructions usually require adjacent registers (based on how registers are named or numbered). Requiring adjacent registers can significantly complicate register allocation and an implementation may chose not to make use of these instructions.

Coding Guidelines

This requirement does more than imply that the **sizeof** operator applied to a complex type will return a value that is exactly twice that returned when the operator is applied to the corresponding real type. It exposes other implementation details that developers might want to make use of. The issues involved are discussed in the following sentence.

Example

```
#include <stdlib.h>

double _Complex *f(void)
{
/*
 * Not allocating an even number of doubles.  Suspicious?
 */
return (double _Complex *)malloc(sizeof(double) * 3);
}
```

506 the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

complex component representation

Commentary

This specification lists additional implementation details that correspond to the Fortran specification. This requirement means that complex types are implemented using Cartesian coordinates rather than polar coordinates. A consequence of the choice of Cartesian coordinates (rather that polar coordinates) is that there are four ways of representing 0 and eight ways of representing infinity (where n represents some value):

+0 + i*0	-0 + i*0	-0 - i*0	+0 - i*0
$+\infty + i*n$	$+\infty + i*\infty$	$n + i*\infty$	$-\infty + i*\infty$
$-\infty + i*n$	$-\infty - i*\infty$	$n - i*\infty$	$+\infty - i*\infty$

The library functions `creal` and `cimag` provide direct access to these components.

C++

Clause 26.2.3 lists the first parameter of the complex constructor as the real part and the second parameter as the imaginary part. But, this does not imply anything about the internal representation used by an implementation.

Other Languages

Fortran specifies the Cartesian coordinate representation.

Coding Guidelines

The standard specifies the representation of a complex type as a two-element array of the corresponding real types. There is nothing implementation-defined about this representation and the guideline recommendation against the use of representation information is not applicable.

569.1 representation information using

One developer rationale for wanting to make use of representation information, in this case, is efficiency. Modifying a single part of an object with complex type invariably involves referencing the other part; for instance, the assignment:

```
val = 4.0 + I * cimag(val);
```

may be considered as too complicated for what is actually involved. A developer may be tempted to write:

```
*(double *)&val = 4.0;
```

as it appears to be more efficient. In some cases it may be more efficient. However, use of the address-of operator is likely to cause translators to be overly cautious, only performing a limited set of optimizations on expressions involving `val`. The result could be less efficient code. Also, the second form creates a dependency on the declared type of `val`. Until more experience is gained with the use of complex types in C, it is not possible to evaluate whether any guideline recommendation is worthwhile.

Example

It is not possible to use a typedef name to parameterize the kind of floating-point type used in the following function.

```
1  double f(double _Complex valu, _Bool first)
2  {
3  double *p_c = (double *)&valu;
4
5  if (first)
6     return *p_c;       /* Real part. */
7  else
8     return *(p_c + 1); /* Imaginary part. */
9  }
```

---

basic types

The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. 507

Commentary

footnote 34 512

This defines the term *basic types* (only used in this paragraph and footnote 34) which was also defined in C90. The term *base types* is sometimes used by developers.

C++

The C++ Standard uses the term *basic types* three times, but never defines it:

3.9.1p10 *[Note: even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types.]*

13.5p7 *The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.*

Footnote 174 *174) An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t` (18.1) and `streamoff` (27.4.1).*

Coding Guidelines

This terminology is not commonly used outside of the C Standard's Committee.

---

types different even if same representation

Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[34)] 508

Commentary

The type checking rules are independent of representation (which can change between implementations). A type is a property in its own right that holds across all implementations. For example, even though the type **char** is defined to have the same range and representation as either of the types **signed char** or **unsigned char**, it is still a different type from them.

char separate type 537

Other Languages

Some languages go even further and specify that all user defined types, even of scalars, are different types. These are commonly called strongly typed languages.

Common Implementations

Once any type compatibility requirements specified in the standard have been checked, implementations are free to handle types having the same representation in the same way. Deleting casts between types having the same representation is so obvious it hardly merits being called an optimization. Some optimizers use type information when performing alias analysis— for instance, in the following definition:

1491 alias analysis

```
1  void f(int *p1, long *p2, int *p3)
2  { /* ... */ }
```

It might be assumed that the objects pointed to by `p1` and `p2` do not overlap because they are pointers to different types, while the objects pointed to by `p1` and `p3` could overlap because they are pointers to the same type.

Coding Guidelines

C does not provide any mechanism for developers to specify that two typedef names, defined using the same integer type, are different types. The benefits of such additional type-checking machinery are usually lost on the C community.

1633 typedef is synonym

Example

```
1  typedef int APPLES;
2  typedef int ORANGES;
3
4  APPLES coxes;
5  ORANGES jafa;
6
7  APPLES totals(void)
8  {
9  return coxes + jafa; /* Adding apples to oranges is suspicious. */
10 }
```

---

509 31) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

footnote 31

Commentary

This interchangeability does not extend to being considered the same for common initial sequence purposes. The sentence that references this footnote does not discuss any alignment issues. This footnote is identical to footnote 39.

1038 common initial sequence

565 footnote 39

Prior to C90 there were no function prototypes. Developers expected to be able to interchange arguments that had signed and unsigned versions of the same integer type. Having to cast an argument, if the parameter type in the function definition had a different signedness, was seen as counter to C's easy-going type-checking system and a little intrusive. The introduction of prototypes did not completely do away with the issue of interchangeability of arguments. The ellipsis notation specifies that nothing is known about the expected type of arguments.

1601 ellipsis supplies no information

Similarly, for function return values, prior to C99 it was explicitly specified that if no function declaration was visible the translator provided one. These implicit declarations defaulted to a return type of **`int`**. If the actual function happened to return the type **`unsigned int`**, such a default declaration might have returned an unexpected result. A lot of developers had a casual attitude toward function declarations. The rest of us have to live with the consequences of the Committee not wanting to break all the source code they wrote. The interchangeability of function return values is now a moot point, because C99 requires that a function declaration be visible at the point of call (a default declaration is no longer provided).

Having slid further down the slippery slope, we arrive at union types. From the efficiency point of view, having to assign a member of a union to another member, having the corresponding (un)signed integer type, knowing that the value is representable, seems overly cautious. If the value is representable in both types, it is a big simplification not to have to be concerned about which member was last assigned to.

This footnote does not explicitly discuss casting pointers to the same signed/unsigned integer type. If objects of these types have the same representation and alignment requirements, which they do, and the value pointed at is within the range common to both types, everything ought to work. However, *meant to imply* does not explicitly apply in this case.

DR #070 *The program is not strictly conforming. Since many pre-existing programs assume that objects with the same representation are interchangeable in these contexts, the C Standard encourages implementors to allow such code to work, but does not require it.*

The program referred to, in this DR, was very similar to the following:

```
1  #include <stdio.h>
2
3  void output(c)
4  int c;
5  {
6  printf("C == %d\n", c);
7  }
8
9  void DR_070(void)
10 {
11 output(6);
12 /*
13  * The following call has undefined behavior.
14  */
15 output(6U);
16 }
```

Other Languages

Few languages support unsigned types as such. Languages in the Pascal family allow subranges to be specified, which could consist of nonnegative values only. However, such subrange types are not treated any differently by the language semantics than when the subrange includes negative values. Consequently, other languages tend to say nothing about the interchangeability of objects having the corresponding signed and unsigned types.

Common Implementations

The standard does not require that this interchangeability be implemented. But it gives a strong hint to implementors to investigate the issue. There are no known implementations that don't do what they are *imply*ed to do.

Coding Guidelines

function declaration use prototype 1810.1 If the guideline recommendation dealing with use of function prototypes is followed, the visible prototype will cause arguments to be cast to the declared type of the parameter. The function return type will also always be known. However, for arguments corresponding to the ellipsis notation, translators will not perform any implicit conversions. If the promoted type of the argument is not compatible with the type that appears in any invocation of the `va_arg` macro corresponding to that argument, the behavior is undefined. Incompatibility between an argument type and its corresponding parameters type (when no prototype is visible) is known to be a source of faults (hence the guideline recommendation dealing with the use of prototypes). So it is to be expected that the same root cause will also result in use of the `va_arg` macro having the same kinds of fault. However, use of the `va_arg` macro is relatively uncommon and for this reason no guideline recommendation is made here.

function declaration use prototype 1810.1

Signed and unsigned versions of the same type may appear as members of union types. However, this footnote does not give any additional access permissions over those discussed elsewhere. Interchangeability of union members is rarely a good idea. 589 union member when written to

What about a pointer-to objects having different signed types? Accessing objects having different types, signed or otherwise, may cause undefined behavior and is discussed elsewhere. The interchangeability being discussed applies to values, not objects. 948 effective type

Example

```
union {
       signed int m_1;
       unsigned int m_2;
      } glob;

extern int g(int, ...);

void f(void)
{
glob.m_2=3;
g(2, glob.m_1);
}
```

---

510 32) See “future language directions” (6.11.1). footnote 32

---

511 33) A specification for imaginary types is in informative annex G. footnote 33

Commentary

This annex is informative, not normative, and is applicable to IEC 60559-compatible implementations. 18 Normative references

C++

There is no such annex in the C++ Standard.

---

512 34) An implementation may define new keywords that provide alternative ways to designate a basic (or any other) type; footnote 34

Commentary

Some restrictions on the form of an identifier used as a keyword are given elsewhere. A new keyword, provided by an implementation as an alternative way of designating one of the basic types, is not the same as a typedef name. Although a typedef name is a synonym for the underlying type, there are restrictions on how it can be used with other type specifiers (it also has a scope, which a keyword does not have). For instance, a vendor may supply implementations for a range of processors and chose to support the keyword **__int_32**. On some processors this keyword is an alternative representation for the type **long**, on others an alternative for the type **int**, while on others it may not be an alternative for any of the basic types. 490 footnote 28
1633 typedef is synonym
1378 type specifier syntax

C90

Defining new keywords that provide alternative ways of designating basic types was not discussed in the C90 Standard.

C++

The object-oriented constructs supported by C++ removes most of the need for implementations to use additional keywords to designate basic (or any other) types

Other Languages

Most languages do not give explicit permission for new keywords to be added to them.

Common Implementations

Microsoft C supports the keyword **__int64**, which specifies the same type as **long long**.

Coding Guidelines

Another difference between an implementation-supplied alternative designation and a developer-defined typedef name is that one is under the control of the vendor and the other is under the control of the developer. For instance, if __int_32 had been defined as a typedef name by the developer, then it would be the developer's responsibility to ensure that it has the appropriate definition in each environment. As an implementation-supplied keyword, the properties of **__int_32** will be selected for each environment by the vendor.

The intent behind supporting new keywords that provide alternative ways to designate a basic type is to provide a mechanism for controlling the use of different types. In the case of integer types the guideline recommendation dealing with the use of a single integer type, through the use of a specific keyword, is applicable here.

object int type only 480.1

Example

```
/*
 * Assume vend_int is a new keyword denoting an alternative
 * way of designating the basic type int.
 */
typedef int DEV_INT;

unsigned DEV_INT glob_1;  /* Syntax violation. */
unsigned vend_int glob_2; /* Can combine with other type specifiers. */
```

---

this does not violate the requirement that all basic types be different. 513

Commentary

The implementation-defined keyword is simply an alternative representation, like trigraphs are an alternative representation of some characters.

Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3. 514

Commentary

footnote 28 490

This sentence duplicates the wording in footnote 28.

---

character types The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. 515

Commentary

This defines the term *character types*.

C++

Clause 3.9.1p1 does not explicitly define the term *character types*, but the wording implies the same definition as C.

Other Languages

Many languages have a character type. Few languages have more than one such type (because they do not usually support unsigned types).

Coding Guidelines

This terminology is not commonly used by developers who sometimes refer to *char types* (plural), a usage that could be interpreted to mean the type **char**. The term *character type* is not immune from misinterpretation either (as also referring to the type **char**). While it does have the advantage of technical correctness, there is no evidence that there is any cost/benefit in attempting to change existing, sloppy, usage.

Table 515.1: Occurrence of character types in various declaration contexts (as a percentage of all character types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
char	16.4	3.6	1.2	0.1	6.6	28.0
signed char	0.2	0.3	0.0	0.1	0.3	1.0
unsigned char	18.1	10.6	0.4	0.8	41.2	71.1
Total	34.7	14.6	1.5	1.0	48.2	

516 The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[35)]

char range, representation and behavior

Commentary

This is a requirement on the implementation. However, it does not alter the fact that the type **char** is a different type than **signed char** or **unsigned char**.

C90

This sentence did not appear in the C90 Standard. Its intent had to be implied from wording elsewhere in that standard.

C++

3.9.1p1

*A **char**, a **signed char**, and an **unsigned char** occupy the same amount of storage and have the same alignment requirements (3.9); that is, they have the same object representation.*

. . .

*In any particular implementation, a plain **char** object can take on either the same values as **signed char** or an **unsigned char**; which one is implementation-defined.*

In C++ the type **char** can cause different behavior than if either of the types **signed char** or **unsigned char** were used. For instance, an overloaded function might be defined to take each of the three distinct character types. The type of the argument in an invocation will then control which function is invoked. This is not an issue for C code being translated by a C++ translator, because it will not contain overloaded functions.

517 An *enumeration* comprises a set of named integer constant values.

enumeration set of named constants

Commentary

There is no phase of translation where the names are replaced by their corresponding integer constant. Enumerations in C are tied rather closely to their constant values. The language has never made the final jump to treating such names as being simply that— an abstraction for a list of names.

Rationale

The C89 Committee considered several alternatives for enumeration types in C:

1. leave them out;

2. include them as definitions of integer constants;
3. include them in the weakly typed form of the UNIX C compiler;
4. include them with strong typing as in Pascal.

The C89 Committee adopted the second alternative on the grounds that this approach most clearly reflects common practice. Doing away with enumerations altogether would invalidate a fair amount of existing code; stronger typing than integer creates problems, for example, with arrays indexed by enumerations.

base document 1

Enumeration types were first specified in a document listing extensions made to the base document.

Other Languages

symbolic name 822

Enumerations in the Pascal language family are distinct from the integer types. In these languages, enumerations are treated as symbolic names, not integer values (although there is usually a mechanism for getting at the underlying representation value). Pascal does not even allow an explicit value to be given for the enumeration names; they are assigned by the implementation. Java did not offer support for enumerated types until version 1.5 of its specification.

Coding Guidelines

symbolic name 822

The benefits of using a name rather than a number in the visible source to denote some property, state, or attribute is discussed elsewhere. Enumerated types provide a mechanism for calling attention to the association between a list (they may also be considered as forming a set) of identifiers. This association is a developer-oriented one. From the translators point of view there is no such association (unlike many other languages, which treat members as belonging to their own unique type). The following discussion concentrates on the developer-oriented implications of having a list of identifiers defined together within the same enumeration definition.

While other languages might require stronger typing checks on the use of enumeration constants and objects defined using an enumerated type, there are no such requirements in C. Their usage can be freely intermixed, with values having other integer types, without a diagnostic being required to be generated. Enumerated types were not specified in K&R C and a developer culture of using macros has evolved. Because enumerated types were not seen to offer any additional functionality, in particular no additional translator checking, that macros did not already provide, they have not achieved widespread usage.

Some coding guideline documents recommend the use of enumerated types over macro names because of the motivation that "using of the preprocessor is poor practice".[798] Other guideline documents specify ways of indicating that a sequence of macro definitions are associated with each other (by, for instance, using comments at the start and end of the list of definitions). The difference between such macro definition usage and enumerations is that the latter has an explicit syntax associated with it, as well as established practices from other languages.

The advantage of using enumerated types, rather than macro definitions, is that there is an agreed-on notation for specifying the association between the identifiers. Static analysis tools can (and do) use this information to perform a number of consistency checks on the occurrence of enumeration constants and objects having an enumerated type in expressions. Without tool support, it might be claimed that there is no practical difference between the use of enumerated types and macro names. Tools effectively enforce stricter type compatibility requirements based on the belief that the definition of identifiers in enumerations can be taken as a statement of intent. The identifiers and objects having a particular enumerated type are being treated as a separate type that is not intended to be mixed with literals or objects having other types.

It is not known whether defining a list of identifiers in an enumeration type rather than as a macro definition affects developer memory performance (e.g., whether developers more readily recall them, their associated properties, or fellow group member names with fewer errors). The issue of identifier naming conventions based on the language construct used to define them is discussed elsewhere

identifier 792
learning a list of
source code 792
context
identifier

The selection of which, if any, identifiers should be defined as part of the same enumeration is based on

concepts that exist within an application (or at least within a program implementing it), or on usage patterns of these concepts within the source code. There are a number of different methods that might be used to measure the extent to which the concepts denoted by two identifiers are similar. The human-related methods of similarity measuring, and mathematical methods based on concept analysis, are discussed elsewhere. Resnick[1158] describes a measure of semantic similarity based on the *is-a* taxonomy that is based on the idea of shared information content.

0 categorization
1821 concept analysis

While two or more identifiers may share a common set of attributes, it does not necessarily mean that they should, or can, be members of the same enumerated type. The C Standard places several restrictions on what can be defined within an enumerated type, including:

- The same identifier, in a given scope, can only belong to one enumeration (Ada allows the same identifier to belong to more than one enumeration in the same scope; rules are defined for resolving the uses of such overloaded identifiers).
- The value of an enumeration constant must be representable in the type **`int`** (identifiers that denote floating-point values or string literals have to be defined as macro names).
- The values of an enumeration must be translation-time constants.

1440 enumeration constant representable in int

Given the premise that enumerated types have an interpretation for developers that is separate from the C type compatibility rules, the kinds of operations supported by this interpretation need to be considered. For instance, what are the rules governing the mixing of enumeration constants and integer literals in an expression? If the identifiers defined in an enumeration are treated as symbolic names, then the operators applicable to them are assignment (being passed as an argument has the same semantics); the equality operators; and, perhaps, the relational operators, if the order of definition has meaning within the concept embodied by the names (e.g, the baud rates that follow are ordered in increasing speed).

The following two examples illustrate how symbolic names might be used by developers (they are derived from the clause on device- and class-specific functions in the POSIX Standard[636]). They both deal with the attributes of a serial device.

- A serial device will have a single data-transfer rate (for simplicity, the possibility that the input rate may be different from the output rate is ignored) associated with it (e.g., its baud rate). The different rates might be denoted using the following definition:

```
enum baud_rates {B_0, B_50, B_300, B_1200, B_9600, B_38400};
```

  where the enumerated constants have been ordered by data-transfer rate (enabling a test using the relational operators to return meaningful information).

- The following definition denotes various attributes commonly found in serial devices:

```
enum termios_c_iflag {
              BRKINT, /* Signal interrupt on break */
              ICRNL,  /* Map CR to NL on input     */
              IGNBRK, /* ignore break condition    */
              IGNCR,  /* Ignore CR                 */
              IGNPAR, /* Ignore characters with parity errors */
              INLCR,  /* Map NL to CR on input     */
              INPCK,  /* Enable input parity check */
              ISTRIP, /* Strip character           */
              IXOFF,  /* Enable start/stop input control  */
              IXON,   /* Enable start/stop output control */
              PARMRK  /* Mark parity errors               */
};
```

where it is possible that more than one of them can apply to the same device at the same time. These enumeration constants are members of a set. Given the representation of enumerations as integer constants, the obvious implementation technique is to use disjoint bit-patterns as the value of each identifier in the enumeration (POSIX requires that the enumeration constants in `termios_c_iflag` have values that are bitwise distinct, which is not met in the preceding definition). The bitwise operators might then be used to manipulate objects containing these values.

The order in which enumeration constants are defined in an enumerated type has a number of consequences, including:

- If developers recognize the principle used to order the identifiers, they can use it to aid recall.
- The extent to which relational operators may be applied.
- Enhancements to the code need to ensure that any ordering is maintained when new members are added (e.g., if a new baud rate, say 4,800, is introduced, should `B_4800` be added between `B_1200` and `B_9600` or at the end of the list?).

The extent to which a meaningful ordering exists (in the sense that subsequent readers of the source would be capable of deducing, or predicting, the order of the identifiers given a description in an associated comment) and can be maintained when applications are enhanced is an issue that can only be decided by the author of the code.

Rev 517.1 When a set of identifiers are used to denote some application domain attribute using an integer constant representation, the possibility of them belonging to an enumeration type shall be considered.

Cg 517.2 The value of an enumeration constant shall be treated as representation information.

Cg 517.3 If either operand of a binary operator has an enumerated type, the other operand shall be declared using the same enumerated type or be an enumeration constant that is part of the definition of that type.

If an enumerated type is to be used to represent elements of a set, it is important that the values of all of its enumeration constants be disjoint. Adding or removing one member should not affect the presence of any other member.

Usage

A study by Gravley and Lakhotia[517] looked at ways of automatically deducing which identifiers, defined as object-like macros denoting an integer constant, could be members of the same enumerated type. The heuristics used to group identifiers were based either on visual clues (block of **`#define`**s bracketed by comments or blank lines), or the value of the macro body (consecutive values in increasing or decreasing numeric sequence; bit sequences were not considered).

macro object-like 1931

The 75 header files analyzed contained 1,225 macro definitions, of which 533 had integer constant bodies. The heuristics using visual clues managed to find around 55 groups (average size 8.9 members) having more than one member, the value based heuristic found 60 such groups (average size 6.7 members).

enumeration different type

Each distinct enumeration constitutes a different *enumerated type*. 518

Commentary

Don't jump to conclusions. Each enumerated type is required to be compatible with some integer type. The C type compatibility rules do not always require two types to be the same. This means that objects declared to have an enumerated type effectively behave as if they were declared with the appropriate, compatible integer type.

1447 enumeration type compatible with

631 compatible type if

C++

The C++ Standard also contains this sentence (3.9.2p1). But it does not contain the integer compatibility requirements that C contains. The consequences of this are discussed elsewhere.

1447 enumeration type compatible with

Other Languages

Languages that contain enumerated types usually also treat them as different types that are not compatible with an integer type (even though this is the most common internal representation used by implementations).

Coding Guidelines

These coding guidelines maintain this specification of enumerations being different enumerated types and recommends that the requirement that they be compatible with some integer type be ignored.

1447 enumeration type compatible with

---

519 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*.

integer types

Commentary

This defines the term *integer types*. Some developers also use the terminology *integral types* as used in the C90 Standard.

C90

In the C90 Standard these types were called either *integral types* or *integer types*. DR #067 lead to these two terms being rationalized to a single term.

C++

> *Types* **bool**, **char**, **wchar_t**, *and the signed and unsigned integer types are collectively called integral types.*[43)] *A synonym for integral type is integer type.*

3.9.1p7

In C the type **_Bool** is an unsigned integer type and `wchar_t` is compatible with some integer type. In C++ they are distinct types (in overload resolution a **bool** or **wchar_t** will not match against their implementation-defined integer type, but against any definition that uses these named types in its parameter list).

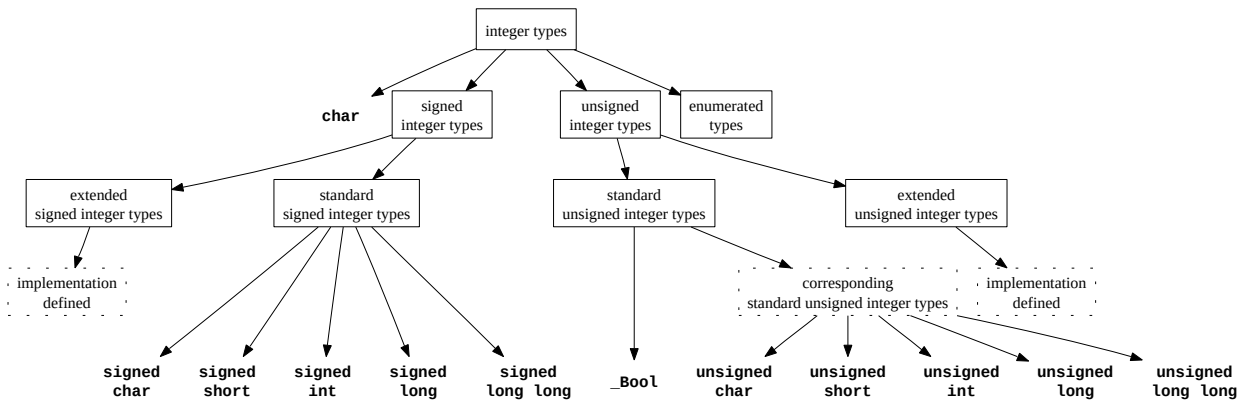


Figure 519.1: The integer types.

standard integer types 493

In C++ the enumerated types are not integer types; they are a compound type, although they may be converted to some integer type in some contexts.

Other Languages

Many other languages also group the character, integer, boolean, and enumerated types into a single classification. Other terms used include *discrete types* and *ordinal types*.

Coding Guidelines

Both of the terms *integer types* and *integral types* are used by developers. Character and enumerated types are not always associated, in developers' minds with this type category.

Table 519.1: Occurrence of integer types in various declaration contexts (as a percentage of those all integer types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	`typedef`	Member	Total
`char`	1.8	0.4	0.1	0.0	0.7	3.1
`signed char`	0.0	0.0	0.0	0.0	0.0	0.1
`unsigned char`	2.0	1.2	0.0	0.1	4.6	7.9
`short`	0.7	0.3	0.0	0.0	0.4	1.4
`unsigned short`	2.3	0.8	0.1	0.1	3.2	6.5
`int`	28.4	10.6	4.2	0.1	6.4	49.7
`unsigned int`	5.6	3.6	0.3	0.1	4.2	13.8
`long`	3.0	1.2	0.1	0.1	0.8	5.1
`unsigned long`	4.8	1.9	0.2	0.1	2.1	9.1
`enum`	0.9	0.9	0.4	0.4	0.8	3.3
Total	49.6	20.8	5.4	0.9	23.2	

---

real types

The integer and real floating types are collectively called *real types*. 520

Commentary

This defines the term *real types*.

C90

C90 did not include support for complex types and this definition is new in C99.

C++

The C++ Standard follows the C90 Standard in its definition of integer and floating types.

Coding Guidelines

This terminology is not commonly used outside of the C Standard. Are there likely to be any guideline recommendations that will apply to real types but not arithmetic types? If there are, then writers of coding guideline documents need to be careful in their use of terminology.

---

arithmetic type

Integer and floating types are collectively called *arithmetic types*. 521

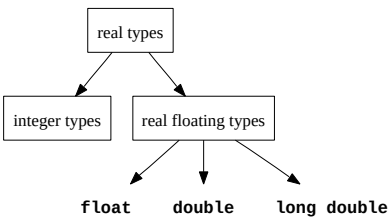


Figure 520.1: The real types.

Commentary

This defines the term *arithmetic types*, so-called because they can appear as operands to the binary operators normally thought of as arithmetic operators.

C90

Exactly the same wording appeared in the C90 Standard. Its meaning has changed in C99 because the introduction of complex types has changed the definition of the term floating types.

497 floating types
three re

C++

The wording in 3.9.1p8 is similar (although the C++ complex type is not a basic type).
The meaning is different for the same reason given for C90.

Coding Guidelines

It is important to remember that pointer arithmetic in C is generally more commonly used than arithmetic on operands with floating-point types (see Table 1154.1, and Table 985.1). There may be coding guidelines specific to integer types, or floating types, however, the category arithmetic type is not usually sufficiently general. Coding guidelines dealing with expressions need to deal with the general, type independent cases first, then the scalar type cases, and finally the more type specific cases.

544 scalar types

Writers of coding guideline documents need to be careful in their use of terminology here. C90 is likely to be continued to be used for several years and its definition of this term does not include the complex types.

---

522 Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

type domain

Commentary

This defines the terms *real type domain* and *complex type domain*. The concept of type domain comes from the mathematics of complex variables. Annex G describes the properties of the *imaginary type domain*. An implementation is not required to support this type domain. Many operations and functions return similar results in both the real and complex domains; for instance:

$$finite / ComplexInf \Rightarrow ComplexZero \tag{522.1}$$
$$finite * ComplexInf \Rightarrow ComplexInf \tag{522.2}$$

However, some operations and functions may behave differently in each domain; for instance:

$$\exp(Inf) \Rightarrow Inf \tag{522.3}$$
$$\exp(-Inf) \Rightarrow 0.0 \tag{522.4}$$
$$\exp(ComplexInf) \Rightarrow ComplexNaN \tag{522.5}$$

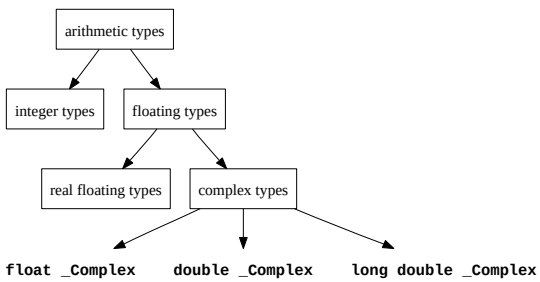


Figure 521.1: The arithmetic types.

Both Inf and $-Inf$ can be viewed as the complex infinity under the Riemann sphere, making the result with an argument of complex infinity nonunique (it depends on the direction of approach to the infinity).

C90

Support for complex types and the concept of type domain is new in C99.

C++

In C++ `complex` is a class defined in one of the standard headers. It is treated like any other class. There is no concept of type domain in the C++ Standard.

Other Languages

While other languages containing a built-in complex type may not use this terminology, developers are likely to use it (because of its mathematical usage).

Coding Guidelines

Developers using complex types are likely to be familiar with the concept of *domain* from their mathematical education.

---

void is incomplete type

The **`void`** type comprises an empty set of values; 523

Commentary

Because types are defined in terms of values they can represent and operations that can be performed on them, the standard has to say what the **`void`** type can represent.

The **`void`** keyword plays many roles. It is the placeholder used to specify that a function returns no value, or that a function takes no parameters. It provides a means of explicitly throwing a value away (using a cast). It can also be used, in association with pointers, as a method of specifying that no information is known about the pointed-to object (pointer to a so-called *opaque* type).

operator () 1000

The use of **`void`** in function return types and parameter definitions was made necessary because nothing appearing in these contexts had an implicit meaning— function returning **`int`** (not supported in C99) and function taking unknown parameters, respectively.

C90

base document 1

The **`void`** type was introduced by the C90 Committee. It was not defined by the base document.

Other Languages

The keyword **`void`** is unique to C (and C++). Some other languages fill the role it plays (primarily in the creation of a generic pointer type) by specifying that no keyword appear. CHILL defines a different keyword, **`PTR`**, for use in this pointer role. Other languages that support a generic pointer type, or have special rules for handling pointers for recursive data structures use concepts that are similar to those that apply to the void type.

Coding Guidelines

generic pointer

The **`void`** type can be used to create an anonymous pointer type or a generic pointer type. The difference between these is intent. In one case there is the desire to hide information and in the other a desire to be able to accept any pointer type in a given context. It can be very difficult, when looking at source code, to tell the difference between these two uses.

coupling 1810

Restricting access to implementation details (through information-hiding) is one way of reducing low-level coupling between different parts of a program. The authors of library functions (either third-party or project-specific) may want to offer a generalized interface to maximize the likelihood of meeting their users' needs without having to provide a different function for every type. Where the calling source code is known, is the use of pointers to **`void`** a lazy approach to passing information around or is it good design practice for future expansion? These issues are higher-level design issues that are outside of the scope of this book.

Usage

Information on keyword usage is given elsewhere (see Table 539.1, Table 758.1, Table 788.1, Table 1003.1, Table 1005.1, and Table 1134.1).

524 it is an incomplete type that cannot be completed.

Commentary

The concept of an incomplete type was not defined in the base document, it was introduced in C90. 1 base document

Defining **void** to be an incomplete type removes the need for lots of special case wording in the standard. A developer defining an object to have the **void** type makes no sense (there are situations where it is of use to the implementation). But because it is an incomplete type, the wording that disallows objects having an incomplete type comes into play; there is no need to introduce extra wording to disallow objects being declared to have the **void** type. Being able to complete the **void** type would destroy the purpose of defining it to be incomplete in the first place. 1818 external linkage exactly one external definition

525 Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows: derived type

Commentary

This defines the term *derived types*. The rules for deciding whether two derived types are compatible are discussed in the clauses for those types.

The translation limits clause places a minimum implementation limit on the complexity of a type and the number of external and block scope identifiers. However, there is no explicit limit on the number of types in a translation unit. Anonymous structure and union declarations, which don't declare any identifiers, in theory consume no memory; a translator can free up all the storage associated with them (but such issues are outside the scope of the standard). 276 translation limits 279 limit type complexity

C++

C++ has derived classes, but it does not define derived types as such. The term *compound types* fills a similar role:

3.9.2p1

> *Compound types can be constructed in the following ways:*

Other Languages

Most languages allow some form of derived types to be built from the basic types predefined by the language. Not all languages support the range of possibilities available in C, while some languages define kinds of derived types not available in C— for instance, sets, tuples, and lists (as built-in types).

Common Implementations

The number of derived types is usually limited by the amount of storage available to the translator. In most cases this is likely to be large.

Coding Guidelines

The term *derived type* is not commonly used by developers. It only tends to crop up in technical discussions involving the C Standard by the Committee.

Derived types are not necessary for the implementation of any application; in theory, an integer type is sufficient. What derived types provide is a mechanism for more directly representing both how an application domain organizes its data and the data structures implied by algorithms (e.g., a linked list) used in implementing an application. Which derived types to define is usually a high-level design issue and is outside the scope of this book. Here we limit ourselves to pointing out constructions that have been known to cause problems in the past.

Table 525.1: Occurrence of derived types in various declaration contexts (as a percentage of all derived types appearing in all of these contexts, e.g., `int **ap[2]` is counted as two pointer types and one array type). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	`typedef`	Member	Total
`*`	30.4	37.6	3.1	0.8	5.6	77.5
array	3.3	0.0	4.4	0.0	3.0	10.8
`struct`	3.7	0.1	2.4	2.3	2.6	11.2
`union`	0.2	0.0	0.0	0.1	0.2	0.5
Total	37.7	37.8	10.0	3.3	11.3	

array type
array
contiguously allocated set of objects

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*.[36)] 526

Commentary

This defines the terms *array type* and *element type*.

Although array types can appear in declarations, they do not often appear as the types of operands. This is because an occurrence, in an expression context, of an object declared to have an array type is often converted into a pointer to its first element. Because of this conversion, arrays in C are often said to be second-class citizens (types). Note that the element type cannot be an incomplete or function type. The standard also specifies a lot of implementation details on how arrays are laid out in storage.

additive 1165
operators
pointer to object

array 994
row-major
storage order

Other Languages

Nearly every language in existence has arrays in one form or another. Many languages treat arrays as having the properties listed here. A few languages simply treat them as a way of denoting a list of locations that may hold values (e.g., Awk and Perl allow the index expression to be a string); it is possible for each element to have a different type and the number of elements to change during program execution. A few languages restrict the element type to an arithmetic type— for instance, Fortran (prior to Fortran 90). The Java reference model does not require that array elements be contiguously allocated. In the case of multidimensional arrays there are implementation advantages to keeping each slice separate (in a garbage collected storage environment it keeps storage allocation requests small).

Coding Guidelines

array type
when to use

The decision to use an array type rather than a structure type is usually based on answers to the following questions:

- Is more than one element of the same type needed?
- Are the individual elements anonymous?
- Do all the elements represent the same applications-domain concept?
- Will individual elements be accessed as a sequence (e.g., indexed with a loop-control variable).

If the individual elements are not anonymous, they might better be represented as a structure type containing two members, for instance, the x and y coordinates of a location. The names of the members also provide a useful aid to remembering what is being represented. In those cases where array elements are used to denote different kinds of information, macros can be used to hide the implementation details. In the following an array holds color and height information. Using macros removes the need to remember which element of the array holds which kind of information. Using enumeration constants is another technique, but it requires the developer to remember that the information is held in an array (and also requires a greater number of modifications if the underlying representation changes):

```
1  #define COLOR(x) (x[0])
2  #define HEIGHT(x) (x[1])
3
4  enum {i_color, i_height};
5
6  extern int abc_info[2];
7
8  void f(void)
9  {
10 int cur_color = COLOR(abc_info);
11 int this_color = abc_info[i_color];
12 }
```

Array types are sometimes the type of choice when sharing data between different platforms (that may use different processors) or between applications written in different languages. The relative position of each element is known, making it easy to code access mechanisms to.

---

527 Array types are characterized by their element type and by the number of elements in the array.

Commentary

This, along with the fact that the first element is indexed from zero, is the complete set of information needed to describe an array type.

989 array subscript identical to

C++

The two uses of the word *characterized* in the C++ Standard do not apply to array types. There is no other similar term applied to array types (8.3.4) in the C++ Standard.

Other Languages

Languages in the Pascal family require developers to specify the lower bound of an array type; it is not implicitly zero. Also the type used to index the array is part of the array type information; the index, in an array access, must have the same type as that given in the array declaration.

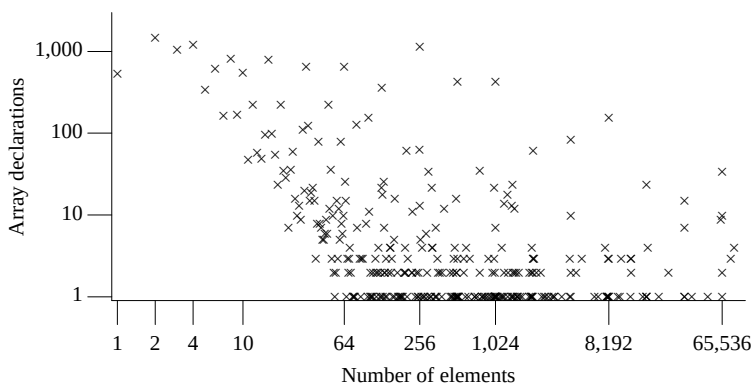


Figure 527.1: Number of arrays defined to have a given number of elements. Based on the translated form of this book's benchmark programs.

Table 527.1: Occurrence of arrays declared to have the given element type (as a percentage of all objects declared to have an array type). Based on the translated form of this book's benchmark programs.

Element Type	%	Element Type	%
`char`	17.2	`struct *`	3.7
`struct`	16.6	`unsigned int`	2.7
`float`	14.6	`enum`	2.5
other-types	10.4	`unsigned short`	2.0
`int`	8.5	`float []`	1.9
`const char`	8.0	`const char * const`	1.3
`char *`	5.1	`short`	1.1
`unsigned char`	4.4		

An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". 528

Commentary

The term *array of T* is the terminology commonly used by developers and is almost universally used in all programming languages.

C++

This usage of the term *derived from* is not applied to types in C++; only to classes. The C++ Standard does not define the term *array of T*. However, the usage implies this meaning and there is also the reference:

3.9p7 *("array of unknown bound of T" and "array of N T")*

The construction of an array type from an element type is called "array type derivation". 529

Commentary

The term *array type derivation* is used a lot in the standard to formalize the process of type creation. It is rarely heard in noncompiler writer discussions.

C++

This kind of terminology is not defined in the C++ Standard.

Other Languages

Different languages use different forms of words to describe the type creation process.

Coding Guidelines

This terminology is not commonly used outside of the C Standard, and there is rarely any need for its use.

structure type sequentially allocated objects

— A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type. 530

Commentary

This defines the term *structure type*. Structures differ from arrays in that their members

- are sequentially allocated, not contiguously allocated (there may be holes, unused storage, between them);
- may have a name;
- are not required to have the same type.

member 1422 address increasing

There are two ways of implementing sequential allocation; wording elsewhere reduces this to one.

C90

Support for a member having an incomplete array type is new in C99.

C++

C++ does not have structure types, it has class types. The keywords **struct** and **class** may both be used to define a class (and *plain old data* structure and union types). The keyword **struct** is supported by C++ for backwards compatibility with C.

Nonstatic data members of a (non-union) class declared without an intervening `access-specifier` *are allocated so that later members have higher addresses within a class object.* 9.2p12

C does not support static data members in a structure, or access-specifiers.

— classes containing a sequence of objects of various types (clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (clause 11); 3.9.2p1

Support for a member having an incomplete array type is new in C99 and not is supported in C++.

In such cases, and except for the declaration of an unnamed bit-field (9.6), the `decl-specifier-seq` *shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.* 7p3

The only members that can have their names omitted in C are bit-fields. Thus, taken together the above covers the requirements specified in the C90 Standard.

Other Languages

Some languages (e.g., Ada and CHILL) contain syntax which allows developers to specify the layout of members in storage (including having relative addresses that differ from their relative textual positions). Languages in the Pascal family say nothing about field ordering. Although most implementations of these languages allocate storage for fields in the order in which they were declared, some optimizers do reorder fields to optimize (either performance, or amount of generated machine code) access to them. Java says nothing about how members are laid out in a class (C structure).

Common Implementations

Some implementations provide constructs that give the developer some control over how members within a structure are laid out. The **#pack** preprocessing directive is a common extension. For example, it may simply indicate that padding between members is to be minimized, or it may take additional tokens to specify the alignments to use. While use of such extensions does not usually affect the type of a structure, developers have to take care that the same types in different translation units are associated with equivalent **#pack** preprocessing directives.

Coding Guidelines

C specifies some of the requirements on the layout of members within a structure, but not all. Possible faults arise when developers make assumptions about member layout which are not guaranteed by the standard (but happen to be followed by the particular implementation they are using). Use of layout information can also increase the effort needed to comprehend source code by increasing the amount of information that needs to be considered during the comprehension process.

Member layout is part of the representation of a structure type and the applicable guideline is the one recommending that no use be made of representation information. In practice developers use representation information to access members in an array-like fashion, and to treat the same area of storage as having different types. 569.1 representation information using

The reason for defining a member as being part of a larger whole rather than an independent object is that it has some form of association with the other members of a structure type. This association may be

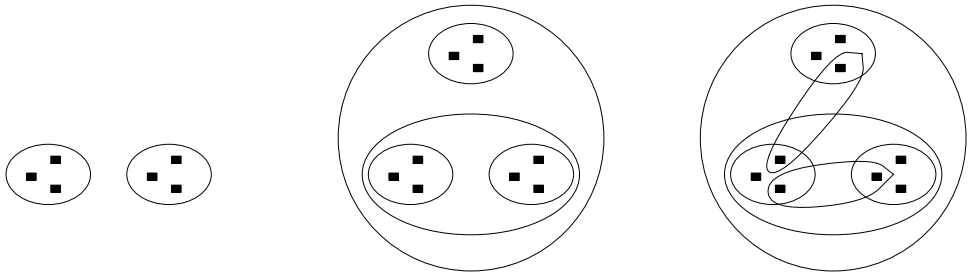


Figure 530.1: Three examples of possible member clusterings. In (a) there are two independent groupings, (b) shows a hierarchy of groupings, while in (c) it is not possible to define two C structure types that share a subset of common member (some other languages do support this functionality). The member c, for instance, might be implemented as a pointer to the value, or it may simply be duplicated in two structure types.

derived from the organization of the application domain, the internal organization of the algorithms used to implement the program, or lower-level implementation details (i.e., if several objects are frequently passed as parameters, or manipulated together the source code can be simplified by passing a single parameter or writing specific functions for manipulating these members). Deciding which structure type (category) a member belongs in is more complicated than the enumeration constant case. Structure types need not be composed of a simple list of members, they can contain instances of other structure types. It is possible to organize structure types into a hierarchy. Organizing the members of structure types is a categorization problem.

categorization 0
enumeration 517
set of named constants
categorization 0

Reorganizing existing structure type declarations to move common member subsets into a new structure definition can be costly (e.g., lots of editing of existing source to change referencing expressions). Your author's experience is that such reorganizations are rarely undertaken. The following are some of the issues that need to be considered in deciding which structure type a member belongs in:

- Is it more important to select a type based on the organization of the application's domain, on the existing internal organization of the source code, or on the expected future organization of the application domain or source code?
- Increasing the nesting of structure definitions will increase the complexity of expressions needed to access some members (an extra selection operator and member name). How much additional cognitive effort is needed to read and comprehend a longer expression containing more member names? This issue is discussed elsewhere.

member selection 1031

- Should the number of members be limited to what is visible on a single screen or printed page? Such a limit implies that the members are somehow related, other than being in the same definition, and that developers would need to refer to different parts of the definition at the same time. If different parts of a definition do need to be viewed at the same time, then comments and blank lines need to be taken into account. It is the number of lines occupied by the definition, not the number of members, that becomes important. The issues involved in laying out definitions are discussed elsewhere. A subset of members sharing an attribute that other members do not have might be candidates for putting in another structure definition.

declaration syntax 1348

- Does it belong to a common subset of members sharing the same attributes and types occurring within two or more structure types? Creating a common type that can be referenced, rather than duplicating members in each structure type, removes the possibility that a change to one of the common members will not be reflected in every type.

Usage

Usage information on the number of members in structure and union types and their types is given elsewhere.

limit 297
members in struct/union
struct 1403
member type

531 — A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

union type
overlapping
members

Commentary

This defines the term *union type*. The members of a union differ from a structure in that they all share the same start address; they overlap in storage. Unions are often used to interpret a storage location in different ways. There are a variety of reasons for wanting to do this, including the next two:

1207 pointer to union members compare equal

- Reducing the number of different objects and functions needed for accessing information in different parts of a program; for instance, it may be necessary to operate on objects having structure types that differ in only a few members. If three different types are defined, it is necessary to define three functions, each using a different parameter type:

```
struct S1 {
           int m1;
           float m2;
           long d1[3];
          };
struct S2 {
           int m1;
           float m2;
           char d2[4];
          };
struct S3 {
           int m1;
           float m2;
           long double d3;
          };

extern void f1(struct S1 *);
extern void f2(struct S2 *);
extern void f3(struct S3 *);
```

If objects operating on these three types had sufficient commonality, it could be worthwhile to define a single type and to reduce the three function definitions to a single, slightly more complicated (it has to work out which member of the union type is currently active) function:

```
struct S4 {
          int m1;
          float m2;
          union {
                long d1[3];
                char d2[4];
                long double d3;
                } m3;
          };

extern void f(struct S4 *);
```

The preceding example relies on the code knowing which union member applies in a given context. A more flexible approach is to use a member to denote the active member of the union:

```
struct node {
            enum {LEAF, UNARY, BINARY, TERNARY} type;
            struct node *left,
                        *right;
            union {
                   struct {
                          char *ident;
```

```
                        } leaf;
                struct {
                        enum {UNARYPLUS, UNARYMINUS} op;
                        struct node *operand;
                        } unary;
                struct {
                        enum {PLUS, MINUS, TIMES, DIVIDE} op;
                        struct node *left,
                                    *right;
                        } binary;
                struct {
                        struct node *test;
                        struct node *iftrue;
                        struct node *iffalse;
                        } ternary;
                } operands;
        };
```

- Another usage is the creation of a visible type punning interface:

```
union {
      int m1; /* Assume int is 24 bits. */
      struct {
              char c1; /* Assumes char is 8 bits and similarly aligned. */
              char c2;
              char c3;
             } m2;
      } x;
```

Here the three bytes of an object, having type **int**, may be manipulated by referencing `c1`, `c2`, and `c3`. The same effect could have been achieved by using pointer arithmetic. In both cases the developer is making use of implementation details, which may vary considerably between implementations.

C++

9.5p1 *Each data member is allocated as if it were the sole member of a struct.*

This implies that the members overlap in storage.

3.9.2p1 *— unions, which are classes capable of containing objects of different types at different times, 9.5;*

7p3 *In such cases, and except for the declaration of an unnamed bit-field (9.6), the* `decl-specifier-seq` *shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.*

The only members that can have their names omitted in C are bit-fields. Thus, taken together the preceding covers the requirements specified in the C Standard.

Other Languages

Pascal does not have an explicit **union** type. However, it does support something known as a *variant record*, which can only occur within a **record** (**struct**) definition. These variant records provide similar, but more limited, functionality (only one variant record can be defined in any record, and it must occur as the last field).

Java's type safety system would be broken if it supported the functionality provided by the C union type and it does not support this form of construct (although the Java library does contain some functions

for converting values of some types, to and from their bit representation— e.g., `floatToIntBits` and `intBitsToFloat`).

The Fortran **EQUIVALENCE** statement specifies that a pair of identifiers share storage. It is possible to equivalence a variable having a scalar type with a particular array element of another object, and even for one array's storage to start somewhere within the storage of a differently named array. This is more flexible than C, which requires that all objects start at the same storage location.

Algol 68 required implementations to track the last member assigned to during program execution. It was possible for the program to do a runtime test, using something called a conformity clause, to find out the member last assigned to.

Coding Guidelines

Union types are created for several reasons, including the following:

- Reduce the number of closely related functions by having one defined with a parameter whose union type includes all of the types used by the corresponding, related functions.
- Access the same storage locations using different types (often to access individual bytes).
- Reduce storage usage by overlaying objects having different types whose access usage is mutually exclusive in the shared storage locations.

Merging closely related functions into a single function reduces the possibility that their functionality will diverge. The extent to which this benefit exceeds the cost of the increase in complexity (and hence maintenance costs) of the single function can only be judged by the developer.

type punning union

Accessing the same storage locations using different types depends on undefined and implementation-defined behaviors. The standard only defines the behavior if the member being read from is the same member that was last written to. Performing such operations is often unconditionally recommended against in coding guideline documents. However, use of a union type is just one of the ways of carrying out type punning. In this case the recommendation is not about the use of union types; it is either about making use of undefined and implementation-defined behaviors, or the use of type punning. (The guideline recommendation on the use of representation information is applicable here.)

589 union member when written to

569.1 representation information using

If developers do, for whatever reason, want to make use of type punning, is the use of a union type better than the alternatives (usually casting pointers)? When a union type is used, it is usually easy to see which different types are involved by looking at the definitions. When pointers are used, it is usually much harder to obtain a complete list of the types involved (all values, and the casts involved, assigned to the pointer object need to be traced). Union types would appear to make the analysis of the code much easier than if pointer types are used.

Using union types to reduce storage usage by overlaying objects having different types is a variation of using the same object to represent different semantic values. The same guideline recommendations are applicable in both cases.

1352 declaration interpretation of identifier

A union type containing two members that have compatible type might be regarded as suspicious. However, the types used in the definition of the members may be typedef names (one of whose purposes is to hide details of the underlying type). Instances of a union type containing two or more members having a compatible type, where neither is a typedef name, are not sufficiently common to warrant a guideline.

Example

```
1  union U_1 {
2              int m_1;
3              int m_2;
4             };
5
6  typedef int APPLES;
```

```
7   typedef int ORANGES;
8
9   union U_2 {
10             APPLES a_count;
11             ORANGES o_count;
12            };
13
14  union U_3 {
15             float f1;
16             unsigned char f_bytes[sizeof(float)];
17            };
```

---

function type — A *function type* describes a function with specified return type. 532

Commentary

function 1821 definition syntax
This defines the term *function type*. A function type is created either by a function definition, the definition of an object or typedef name having type pointer-to function type, or a function declarator. function 1592 declarator return type

According to this definition, the parameters are not part of a function's type. The type checking of the arguments in a function call, against the corresponding declaration, are listed as specific requirements under the function call operator rather than within the discussion on types. However, the following sentence includes parameters as part of the characterization of a function type. function call 997

C++

3.9.2p1 — *functions, which have parameters of given types and return* ***void*** *or references or objects of a given type, 8.3.5;*

The parameters, in C++, need to be part of a function's type because they may be needed for overload resolution. This difference is not significant to developers using C because it does not support overloaded functions.

Other Languages

Functions, as types, are not common in other algorithmic languages. Although nearly every language supports a construct similar to C functions, and may even call them function (or procedure) types, it is not always possible to declare objects to refer to these types (like C supports pointers to functions). Some languages do allow references to function types to be passed as arguments in function calls (this usage invariable requires some form of prototype definition). Then the called function is able to call the function referred to by the corresponding parameter. In functional languages, function types are an integral part of the design of the language type system. A Java method could be viewed as a function type.

A function type is characterized by its return type and the number and types of its parameters. 533

Commentary

function 1522 specifier syntax
The **inline** function specifier is not part of the type.

C++

C++ defines and uses the concept of function *signature* (1.3.10), which represents information amount the number and type of a function's parameters (not its return type). The two occurrences of the word *characterizes* in the C++ Standard are not related to functions.

Usage

Usage information on function return types is given elsewhere (see Table 1005.1) as is information on parameters (see Table 1831.1).

534 A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*".

function returning T

Commentary

The term *function returning T* is a commonly used term. The term *function taking parameters* is heard, but not as often.

C++

The term *function returning T* appears in the C++ Standard in several places; however, it is never formally defined.

Other Languages

Different languages use a variety of terms to describe this kind of construct.

535 The construction of a function type from a return type is called "function type derivation".

Commentary

This terminology may appear in the standard, but it is very rarely heard in discussions.

C++

There is no such definition in the C++ Standard.

536 35) **CHAR_MIN**, defined in **<limits.h>**, will have one of the values 0 or **SCHAR_MIN**, and this can be used to distinguish the two options.

footnote 35

Commentary

Similarly, CHAR_MAX will have one of two values that could be used to distinguish the two options.

312 CHAR_MAX
326 char
if treated as signed integer

C++

The C++ Standard includes the C90 library by reference. By implication, the preceding is also true in C++.

Example

```
#include <limits.h>

#if CHAR_MIN == 0
/* ... */
#elif CHAR_MIN == SCHAR_MIN
/* ... */
#else
#error Broken implementation
#endif
```

537 Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either.

char separate type

Commentary

This sentence calls out a special case of the general rule that any two basic types are different. In most cases, **char** not being compatible with its *matching* type is not noticeable (an implicit conversion is performed). However, while objects having different character types may be assigned to each other, a pointer-to **char** and a pointer to any other character type may not.

508 types different
even if same representation

C++

3.9.1p1

Plain **char**, **signed char**, *and* **unsigned char** *are three distinct types.*

Common Implementations

Some implementations have been known to not always honor this requirement, treating **char** and its *matching* character type as if they were the same type.

Coding Guidelines

A common developer expectation is that the type **char** will be treated as either of the types **signed char**, or an **unsigned char**. It is not always appreciated that the types are different. Apart from the occasional surprise, this incorrect assumption does not appear to have any undesirable consequences.

Example

```
1  char p_c,
2       *p_p_c = &p_c;
3  signed char s_c,
4              *p_s_c = &s_c;
5  unsigned char u_c,
6                *p_u_c = &u_c;
7
8  void f(void)
9  {
10 p_c = s_c;
11 p_c = u_c;
12
13 p_p_c = p_s_c;
14 p_p_c = p_u_c;
15 }
```

---

footnote 36

36) Since object types do not include incomplete types, an array of incomplete type cannot be constructed. 538

Commentary

Object types do not include function types either. But they do include pointer-to function types.

C++

3.9p6 *Incompletely-defined object types and the void types are incomplete types (3.9.1).*

object types 475 The C++ Standard makes a distinction between incompletely-defined object types and the **void** type.

3.9p7 *The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type.*

The following deals with the case where the size of an array may be omitted in a declaration:

8.3.4p3 *When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence.*

Arrays of incomplete structure and union types are permitted in C++.

```
{
struct st;
typedef struct st_0 A[4]; /* Undefined behavior */
                          // May be well- or ill-formed
typedef struct st_1 B[4]; /* Undefined behavior */
                          // May be well- or ill-formed
struct st_0 {             /* nothing has changed */
              int mem;    // declaration of A becomes well-formed
             };
}                         /* nothing has changed */
                          // declaration of B is now known to be ill-formed
```

Other Languages

Most languages require that the size of the element type of the array be known at the point the array type is declared.

---

539 — A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*.

pointer type
referenced type

Commentary

This defines the terms *pointer type* (a term commonly used by developers) and *referenced type* (a term not commonly used by C developers). Pointers in C have maximal flexibility. They can point at any kind of type (developers commonly use the term *point at*).

C++

C++ includes support for what it calls *reference* types (8.3.2), so it is unlikely to use the term *referenced type* in this context (it occurs twice in the standard). There are requirements in the C++ Standard (5.3.1p1) that apply to pointers to object and function types, but there is no explicit discussion of how they might be created.

Other Languages

Some languages do not include pointer types. They were added to Fortran in its 1991 revision. They are not available in Cobol. Some languages do not allow pointers to refer to function types, even when the language supports some form of function types (e.g., Pascal). Java does not have pointers, it has references.

Coding Guidelines

Use of objects having pointer types is often considered to be the root cause of many faults in programs written in C. Some coding guideline documents used in safety-critical application development prohibit the use of pointers completely, or severely restrict the operations that may be performed on them. Such prohibitions are not only very difficult to enforce in practice, but there is no evidence to suggest that the use of alternative constructs reduces the number of faults.

String literals have type pointer to **char**; an array passed as an argument will be converted to a pointer to its element type, and without the ability to declare parameters having a pointer type, significantly more information has to be passed via file scope objects (pointers are needed to implement the parameter-passing concept of call-by address).

903 string literal
static storage duration

Table 539.1: Occurrence of objects declared using a given pointer type (as a percentage of all objects declared to have a pointer type). Based on the translated form of this book's benchmark programs.

Pointed-to Type	%	Pointed-to Type	%
`struct`	66.5	`struct *`	1.8
`char`	8.0	`int`	1.8
`union`	6.0	`const char`	1.3
other-types	5.5	`char *`	1.2
`void`	3.3	str | str	
`unsigned char`	2.6	`_double` | `_double`	
`unsigned int`	2.2	`_double` | `_double`	

pointer type describes a

A pointer type describes an object whose value provides a reference to an entity of the referenced type. 540

Commentary

The value in a pointer object is a reference. Possible implementations of a reference include an address in storage, or an index into an array that gives the actual address. In C the value of a pointer object is rarely called a *reference*. The most commonly used terminology is *address*, which is what a pointer value is on most implementations. Sometimes the term *pointed-to object* is used.

Other Languages

Other languages use a variety of terms to refer to the value stored in an object having pointer type. Java uses the term *reference* exclusively; it does not use the term *pointer* at all. The implementation details behind a Java reference are completely hidden from the developer.

Common Implementations

pointer 590 segmented architecture

Nearly every implementation known to your author represents a reference using the address of the referenced object only (in some cases the address may not be represented as using a single value). Some implementations use a, so-called, *fat pointer* representation.[65, 1288] These implementations are usually intended for use during program development, where information on out-of-bounds storage accesses is more important than speed of execution. A fat pointer includes information on the pointed-to object such as its base address and the number of bytes in the object.

In the Model Implementation C Checker[681] a function address is implemented as two numbers. One is an index into a table specifying the file containing the translated function definition and the other is an index into a table specifying the offset of the executable code within that file. The IBM AIX compiler[618] also uses a function descriptor, not the address of the generated code.

The Java virtual machine does not include the concept of pointer (or address). It includes the concept of a reference; nothing is said about how such an entity is implemented. A C translator targeted at this host would have to store a JVM reference in an object having pointer type.

Some processors differentiate between different kinds of main storage. Access to different kinds of storage are faster/slower, or accesses to particular storage areas may only be made via particular registers. Translators for such processors usually provide keywords, enabling developers to specify which kinds of storage pointers will be pointing at.

Some implementations use different pointer representations (they usually vary in the number of bytes used), depending on how the pointer is declared. For instance, the keywords **`near`**, **`far`**, and **`huge`** are sometimes provided to allow developers to specify the kind of representation to use.

The HP C/iX translator[1039] supports a short pointer (32-bit) and long pointer (64-bit). A long pointer has two halves, 32 bits denoting a process-id and 32 bits denoting the offset within that process address space. The Unisys A Series implementation[1390] represents pointers as integer values. A pointer to **`char`** is the number of bytes from the start of addressable storage (for that process), not the logical address of the storage location. A pointer to the types **`int`** and **`float`** is the number of words (6 bytes) from the start of storage (the unit of measurement for other types is the storage alignment used for the type).

Zhang and Gupta[1511] developed what they called the *common prefix* transformation, which compresses a 32-bit pointer into 15 bits (this is discussed elsewhere). There has been some research[1311] investigating the use of Gray codes, rather than binary, to represent addresses. Successive values in a Gray code differ from each other in a single bit. This property can have advantages in high-performance, or low-power (electrical) situations when sequencing through a series of storage locations.

1422 pointer compressing members

---

541 A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*".

Commentary

The term *pointer to T* is commonly used by developers, and is almost universally used for all programming languages.

Other Languages

This term is almost universally used for all languages that contain pointer types.

---

542 The construction of a pointer type from a referenced type is called "pointer type derivation".

Commentary

The term *pointer type derivation* is used a lot in the standard to formalize the process of type creation. It is rarely heard outside of the committee and compiler writer discussions.

C++

The C++ Standard does not define this term, although the term *derived-declarator-type-list* is defined (8.3.1p1).

Other Languages

Different languages use different forms of words to describe the type creation process.

---

543 These methods of constructing derived types can be applied recursively.

Commentary

There are two methods of constructing derived types in the visible source; either a typedef name can be used, or the declaration of the type can contain nested declarations. For instance, an array of array of *type T* might be declared as follows:

```
1  typedef int at[10];
2  at obj_x[5];
3
4  int obj_y[10][5];
```

The standard specifies minimum limits on the number of declarators that an implementation is required to support.

279 limit type complexity

Other Languages

Most languages support more than one level of type derivation. Many languages support an alternative method of declaring multidimensional arrays, where **[ ]** are not treated as an operator. Structure types were added in Fortran 90, and only support a single nesting level.

1577 footnote 121

---

544 Arithmetic types and pointer types are collectively called *scalar types*.

scalar types

Commentary

This defines the term *scalar type*. It is commonly used by developers and it is also used in many other programming languages. The majority of operations in C act on objects and values having a scalar type.

C++

3.9p10 *Arithmetic types (3.9.1), enumeration types, pointer types, and pointer to member types (3.9.2), and* `cv-qualified` *versions of these types (3.9.3) are collectively called scalar types.*

While C++ includes type qualifier in the definition of scalar types, this difference in terminology has no impact on the interpretation of constructs common to both languages.

Other Languages

The term *scalar type* is used in many other languages. Another term that is sometimes heard is *simple type*.

Common Implementations

Many processors only contain instructions that can operate on values having a scalar type. Operations on aggregate types are broken down into operations on their constituent scalar components. Many implementations only perform some optimizations at the level of scalar types (components of derived types having a scalar type are considered for optimization, but the larger whole is not). For instance, the level of granularity used to allocate values to registers is often at the level of scalar types.

Coding Guidelines

Pointer types commonly occur in many of the same contexts as arithmetic types. Having guideline recommendations that apply to both is often a useful generalization and reduces the number of special cases. The following is a meta-guideline recommendation.

Rev 544.1 Where possible coding guidelines shall try to address scalar types, rather than just arithmetic types.

---

aggregate type Array and structure types are collectively called *aggregate types*.[37)] 545

Commentary

This defines the term *aggregate type*. This terminology is often incorrectly used by developers. An aggregate type includes array types but does not include union types.

C++

8.5.1p1 *An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).*

Class types in C++ include union types. The C definition of aggregate does not include union types. The difference is not important because everywhere that the C++ Standard uses the term *aggregate* the C Standard specifies aggregate and union types.

The list of exclusions covers constructs that are in C++, but not C. (It does not include static data members, but they do not occur in C and are ignored during initialization in C++.) There is one place in the C++ Standard (3.10p15) where the wording suggests that the C definition of aggregate is intended.

Coding Guidelines

Although the logic behind the term *aggregate type* is straight-forward, a type made up of more than one object (the array type having one element, or the structure type having one member, is considered to be a degenerate case), is a categorization of types that is rarely thought about by developers. In most developer discussions, array and structure types are not thought of as belonging to a common type category. type category 553

The term *aggregate type* is commonly misused. Many developers assume that it includes the union types in its definition; they are not aware that array types are included in the definition. To avoid confusion, this term is probably best avoided in coding guideline documents.

546 An array type of unknown size is an incomplete type.

array unknown size

Commentary

The size is unknown in the sense that the number of elements is not known. The term *incomplete array type* is often used. Objects with no linkage cannot have an incomplete type.

1573 array incomplete type
1361 object type complete by end

Other Languages

Some languages include the concept of an array having an unknown number of elements. This usually applies only to the type of a parameter in which arrays with different numbers of elements are passed. There is often a, language-provided, mechanism for finding the number of elements in the array actually passed as an argument. In other cases it is the developer's responsibility to pass that information as an argument, along with the array itself. In Java all declarations of objects having an array type omit the number of elements. The actual storage is allocated during program execution using the operator **new**.

547 It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage).

array type completed by

Commentary

A typedef name that has an incomplete array type cannot be completed. However, an object definition, whose type specifier is the typedef name, can complete this type for its definition. An initializer appearing as part of the object's definition provides a mechanism for a translator to deduce the size of the array type. The size, actually the number of elements, may also be implicitly specified, if there is no subsequent declaration that completes the type, when the end of the translation unit is reached.

In the case of parameters their type is converted to a pointer to the element type.

1683 array of unknown size initialized
1850 object definition implicit
1598 array type adjust to pointer to

C++

> *The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on;* 3.9p7

Which does not tell us how it got completed. Later on in the paragraph we are given the example:

```
extern int arr[];   // the type of arr is incomplete

int arr[10];        // now the type of arr is complete
```

3.9p7

which suggests that an array can be completed, in a later declaration, by specifying that it has 10 elements. :-)

Other Languages

Fortran supports the declaration of subroutine parameters taking array types, whose size is not known at translation time. Array arguments are passed by reference, so the translator does not need to know their size. Developers usually pass the number of elements as another parameter to the subroutine.

548 A type has *knownconstantsize* if the type is not incomplete and is not a variable length array type.

known constant size

Commentary

The sentence was added by the response to DR #312 and clarifies that *known constant size* is to be interpreted as a technical term involving types and not the kind of expressions that an implementation may chose to treat as being constants.

822 constant syntax

549 A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type.

structure incomplete type union incomplete type

Commentary

type 1454 contents defined once

Incomplete structure and union types are needed to support self-recursive and mutually recursive declarations involving more than one such structure or union. These are discussed in subclause 6.7.2.3.

Other Languages

A mechanism for supporting mutual recursion in type definitions is invariably provided in languages that support some form of pointer type. A variety of special rules is used by different languages to allow mutually referring data types to be defined.

Example

```
1  struct U {
2            int M1;
3            struct U *next;
4           };
5
6  struct S;  /* Members defined later. */
7
8  struct T {
9            int m1;
10           struct S *s_m;
11          };
12 struct S { /* S now completed. */
13           long m1;
14           struct T *t_m;
15          };
```

The two mutually referential structure types could not be declared without the original, incomplete declaration of S.

---

incomplete type completed by

It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope. 550

Commentary

A definition of the same tag name in a different scope is a different definition and does not complete the declaration in the outer scope.

C++

3.9p7 *A class type (such as "`class X`") might be incomplete at one point in a translation unit and complete later on;*

An example later in the same paragraph says:

3.9p7

```
class X;                // X is an incomplete type

struct X { int i; };    // now X is a complete type
```

The following specifies when a class type is completed; however, it does not list any scope requirements.

9.2p2 *A class is considered a completely-defined object type (3.9) (or complete type) at the closing `}` of the `class-specifier`.*

In practice the likelihood of C++ differing from C, in scope requirements on the completion of types, is small and no difference is listed here.

Coding Guidelines

1707 statements integrating information between

Having important references to an identifier close together in the visible source code has a number of benefits. In the case of mutually recursive structure and union types, this implies having the declarations and definitions adjacent to each other.

Rev 550.1 The completing definition, of an incomplete structure or union type, shall occur as close to the incomplete declaration as permitted by the rules of syntax and semantics.

Example

```
1  struct INCOMP_TAG;              /* First type declaration. */
2  struct INCOMP_TAG *gp;          /* References first type declaration. */
3  extern void f(struct INCOMP_TAG /* Different scope. */);
4
5  void g(void)
6  {
7  struct INCOMP_TAG {int mem1;} *lp; /* Different type declaration. */
8
9  lp = gp; /* Not compatible pointers. */
10 }
11
12 void h(void)
13 {
14 struct INCOMP_TAG; /* Different type declaration. */
15 struct INCOMP_TAG *lp;
16
17 lp = gp; /* Not compatible pointers. */
18 }
19
20 struct INCOMP_TAG {
21                     int m_1;
22                    };
```

---

551 Array, function, and pointer types are collectively called *derived declarator types*.

derived declarator types

Commentary

This defines the term *derived declarator types*. This term is used a lot in the standard to formalize the process of type creation. It is rarely heard outside of committee and translator writer discussions.

C++

There is no equivalent term defined in the C++ Standard.

Other Languages

Different languages use different terms to describe the type creation process.

Coding Guidelines

This terminology is not commonly used outside of the C Standard and its unfamiliarity, to developers, means there is little to be gained by using it in coding guideline documents.

---

552 A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

Commentary

This defines the term *declarator type derivation*. This term does not appear to be used anywhere in the standard, except the index and an incorrect forward reference.

C++

There is no equivalent definition in the C++ Standard, although the description of compound types (3.9.2) provides a superset of this definition.

---

type category

A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types. 553

Commentary

This defines the term *type category*. Other terms sometimes also used by developers, which are not defined in the standard, are *outermost type* and *top level type*. An object is commonly described as an *array-type*, a *pointer-type*, a *structure-type*, and so on. Without reference to its constituents. But the term *type category* is rarely heard in developer discussions.

The following was included in the response to DR #272:

DR #272

Committee Discussion *(for history only)*

The committee wishes to keep the term "type category" for now, removing the term "type category" from the next revision of the standard should be considered at that time.

C++

The term *outermost level* occurs in a few places in the C++ Standard, but the term *type category* is not defined.

Other Languages

The concept denoted by the term *type category* exists in other languages and a variety of terms are used to denote it.

Coding Guidelines

The term *type category* is not commonly used by developers (it only occurs in five other places in the standard). Given that terms such as *outermost type* are not commonly used either, it would appear that there is rarely any need to refer to the concept denoted by these terms. Given that there is no alternative existing common practice there is no reason not to use the technically correct term; should a guidelines document need to refer to this concept.

---

unqualified type

Any type so far mentioned is an *unqualified type*. 554

Commentary

This defines the term *unqualified type*. An unqualified type is commonly referred to, by developers, as just the *type*, omitting the word unqualified. The suffix qualified is only used in discussions involving type qualifiers, to avoid ambiguity.

C++

In C++ it is possible for the term *type* to mean a qualified or an unqualified type (3.9.3).

Coding Guidelines

It is common practice to use the term *type* to mean the unqualified type. Unqualified types are much more commonly used than qualified types. While the usage of the term *type* might be generally accepted by C developers to mean an unqualified type, this usage is not true in C++. Guidelines that are intended to be applied to both C and C++ code will need to be more precise in their use of terminology than if they were aimed at C code only.

Table 554.1: Occurrence of qualified types as a percentage of all (i.e., qualified and unqualified) occurrences of that kind of type (e.g., * denotes any pointer type, **struct** any structure type, and *array of* an array of some type). Based on the translated form of this book's benchmark programs.

Type Combination	%	Type Combination	%
array of **const**	26.7	**const** *	0.4
const integer-type	4.8	**const union**	0.3
const real-type	2.7	**volatile struct**	0.1
* **const**	2.6	**volatile** integer-type	0.1
const struct	2.4	* **volatile**	0.1

555 Each unqualified type has several *qualified versions* of its type,[38] corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. qualified type versions of

Commentary

This defines the term *qualified version* of a type. In the case of structure, and union types, qualifiers also qualify their member types. Type qualifiers provide a means for the developer to provide additional information about the properties of an object. In general these properties relate to issues involved with the optimization of C programs.

C90

The **noalias** qualifier was introduced in later drafts of what was to become C90. However, it was controversial and there was insufficient time available to the Committee to resolve the issues involved. The noalias qualifier was removed from the document, prior to final publication. The **restrict** qualifier has the same objectives as **noalias**, but specifies the details in a different way.

Support for the **restrict** qualifier is new in C99.

C++

Each type which is a cv-unqualified complete or incomplete object type or is **void** *(3.9) has three corresponding cv-qualified versions of its type: a* `const-qualified` *version, a* `volatile-qualified` *version, and a* `const-volatile-qualified` *version.* 3.9.3p1

The **restrict** qualifier was added to C99 while the C++ Standard was being finalized. Support for this keyword is not available in C++.

Other Languages

Pascal uses the **packed** keyword to indicate that the storage occupied by a given type should be minimized (packed, so there are no unused holes). Java has 10 different modifiers; not all of them apply directly to types. Some languages contain a keyword that enables an object to be defined as being read-only.

Common Implementations

The standard provides a set of requirements that an implementation must honor for an object with a given qualified type. The extent to which a particular translator makes additional use of the information provided varies.

Table 555.1: Occurrence of type qualifiers on the outermost type of declarations occurring in various contexts (as a percentage of all type qualifiers on the outermost type in these declarations). Based on the translated form of this book's benchmark programs.

Type Qualifier	Local	Parameter	File Scope	**typedef**	Member	Total
const	18.5	4.3	50.8	0.0	1.2	74.8
volatile	1.6	0.1	3.0	0.1	20.4	25.2
volatile const	0.0	0.0	0.0	0.0	0.0	0.0
Total	20.1	4.4	53.8	0.1	21.6	

qualifiers representation and alignment

The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[39)] 556

Commentary

Qualifiers apply to objects whose declarations include them. They do not play any part in the interpretation of a value provided by a type, but they participate in the type compatibility rules.

Other Languages

This statement can usually be applied to qualifiers defined in other languages.

Common Implementations

Objects declared using a qualified type may have the same representation and alignment requirements, but there are no requirements specifying where they might be allocated in storage. Some implementations chose to allocate differently qualified objects in different areas of storage. For instance, const-qualified objects may be placed in read-only storage; volatile-qualified objects may be mapped to special areas of storage associated with I/O ports.

derived type qualification

A derived type is not qualified by the qualifiers (if any) of the type from which it is derived. 557

Commentary

For instance, a type denoting a const-qualified **char** does not also result in a pointer to it to also being const-qualified, although the pointed-to type retains its **const** qualifier.

A structure type containing a member having a qualifier type does not result in that type also being so qualified. However, an object declared to have such a structure type will share many of the properties associated with objects having the member's qualified type when treated as a whole. For instance, the presence of a member having a const-qualified type, in a structure type, prevents an object declared using it from appearing as the left operand of an assignment operator. However, the fact that one member has a const-qualified type does not affect the qualification of any other members of the same structure type.

Other Languages

This specification usually applies to other languages that support some form of type qualifiers, or modifiers.

Coding Guidelines

Inexperienced developers sometimes have problems distinguishing between constant pointers to types and pointers to constant types. Even the more experienced developer might be a little confused over the following being conforming:

```
1  void f(int * const a[])
2  {
3  a++; /* Type of a is pointer to constant pointer to int. */
4  }
```

Rev 557.1 Array and pointer types that include a qualifier shall be checked to ensure that the type that is so qualified is the one intended by the original author.

Translators will probably issue a diagnostic for those cases in which a **const** qualifier was added where it was not intended (e.g., because of an attempt to modify a value). However, translators are not required to issue a diagnostic for a **const** qualifier that has been omitted (unless there is a type compatibility associated with the assignment, or argument passing). Some static analysis tools[436,438] diagnose declarations where a **const** qualifier could be added to a type without violating any constraints.

Example

In the following declarations x1 is not a const-qualified structure type. However, one of its members is const-qualified. The member x1.m2 can be modified. y1 is a const-qualified structure type. The member y1.m2 cannot be modified.

```
typedef const int CI;

CI *p;   /* The pointed-to type is qualified, not the pointer. */
CI a[3]; /*
          * a is made up of const ints, it is not
          * possible to qualify the array type.
          */

struct S1 {
            const int m1;
            long m2;
           } x1, x2;

const struct S2 {
                  const int m1;
                  long m2;
                 } y1;

void f(void)
{
x1 = x2; /* Constraint violation. */
}
```

What are the types in:

```
typedef int *I;

I const p1; /* A const qualified pointer to int. */
const I p2; /* A const qualified pointer to int. */
```

---

558 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[39)]

pointer to void same representation and alignment as

Commentary

This is a requirement on the implementation. In its role as a generic container for any pointer value, a pointer to **void** needs to be capable of holding the *hardest* reference to represent. Experience has shown that, in those cases where different representations are used for pointers to different types, this is usually the pointer to character type.

Prior to the publication of C90, pointers to character types were often used to perform the role that pointer to **void** was designed to fill. That is, they were the pointer type used to represent the concept of pointer to any type, a generic pointer type (through suitable casting, which is not required for pointer to **void**). Existing code that uses pointer to character type as the generic pointer type can coexist with newly written code that uses pointer to **void** for this purpose.

523 generic pointer

Other Languages

Most languages that contain pointer types do not specify a pointer type capable of representing any other pointer type. Although pointer to character type is sometimes used by developers for this purpose.

Coding Guidelines

This C requirement is intended to allow existing code to coexist with newly written code using pointer to **void**. Mixing the two pointer types in newly written code serves no useful purpose. The fact that the two

kinds of pointers have the same representation requirements does not imply that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

representation information using 569.1

Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. 559

pointer to qualified/unqualified types

Commentary

This is a requirement on the implementation.

The representation and alignment of a type is specified as being independent of any qualifiers that might appear on the type. Since the pointed-to type has these properties, it might be expected that pointers to them would also have these properties.

qualifiers 556 representation and alignment

Common Implementations

This requirement on the implementation rules out execution-time checking of pointer usage by using different representations for pointers to qualified and unqualified types.

banked storage

The C model of storage is of a flat (in the sense of not having any structure to it) expanse into which objects can be allocated. Some processors have disjoint storage areas (or *banks*). They are disjoint in that either different pointer representations are required to access the different areas, or because execution of a special instruction causes subsequent accesses to reference a different storage area. The kind of storage referred to by a pointer value, may be part of the encoding of that value, or the processor may have state information that indicates which kind of storage is currently the default to be accessed, or the kind of storage to access may be encoded in the instruction that performs the access.

The IAR PICmicro compiler[612] provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size.

Coding Guidelines

The fact that the two kinds of pointers have the same representation requirements does not imply that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

representation information using 569.1

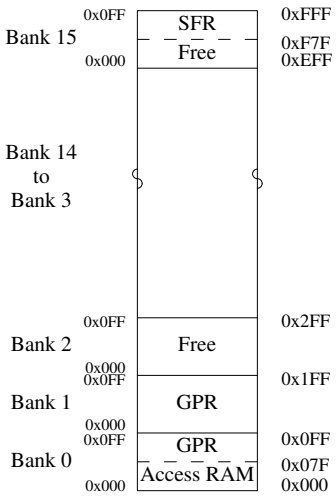


Figure 559.1: Data storage organization for the PIC18CXX2 devices[931] The 4,096 bytes of storage can be treated as a linear array or as 16 banks of 256 bytes (different instructions and performance penalties are involved). Some storage locations hold Special Function Registers (SFR) or General Purpose Registers (GPR). *Free* denotes storage that does not have a preassigned usage and is available for general program use.

560 All pointers to structure types shall have the same representation and alignment requirements as each other.

alignment pointer to structures representation pointer to structures

Commentary

This is a requirement on the implementation. It refers to the pointer type, not the pointed-to type. This specification is redundant in that it can be deduced from other requirements in the standard. A translation unit can define a pointer to an incomplete type, where no information on the pointed-to type is provided within that translation unit. In:

```
#include <stdlib.h>

extern struct tag *glob_p;

int f(void)
{
if (glob_p == NULL)
   return 1;

glob_p = (struct tag *)malloc(8);
return 0;
}
```

a translator knows nothing about the pointed-to type (apart from its tag name, and it would be an unusual implementation that based alignment decisions purely on this information). If pointers to different structure types had different representations and alignments, the implementation would have to delay generating machine code for the function `f` until link-time.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

Other Languages

Most languages do not get involved in specifying details of pointer representation and alignment.

Coding Guidelines

The fact that the two kinds of pointers have the same representation requirements does not mean that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

569.1 representation information using

561 All pointers to union types shall have the same representation and alignment requirements as each other.

alignment pointer to unions representation pointer to unions

Commentary

The chain of deductions made for pointers to structure types also apply to pointer-to union types.

560 alignment pointer to structures

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

Other Languages

Most languages do not get involved in specifying details about pointer representation and alignment.

Coding Guidelines

The fact that the two kinds of pointers have the same representation requirements does not mean that they represent a reference to the same object with the same pattern of bits (any more than two pointers of the same type are required to). The guideline recommendation dealing with the use of representation information is applicable here.

representation information using 569.1

---

alignment pointers

Pointers to other types need not have the same representation or alignment requirements. 562

Commentary

Although many host processors use the same representation for all pointer types, this is not universally true, and this permission reflects this fact.

C++

3.9.2p3 *The value representation of pointer types is implementation-defined.*

Other Languages

Most languages do not get involved in specifying details of pointer representation and alignment.

Common Implementations

word addressing 54

Some processors use what is sometimes known as word *addressing*. This hardware characteristic may, or may not, result in some pointer types having different representations.

---

footnote 37

37) Note that aggregate type does not include union type because an object with union type can only contain one member at a time. 563

Commentary

As a phrase, the term *aggregate type* is open to several interpretations. Experience shows that developers sometimes classify union types as being aggregate types. This thinking is based on the observation that structure and union types often contain many different types— an aggregate of types. However, the definition used by the Committee is based on there being an aggregate of objects. Although an object having a union type can have many members, only one of them represents a value at any time (an object having a structure or array type is usually capable of representing several values at the same time).

The phrase *one member at a time* is a reference to the fact that the value of at most one member can be stored in an object having a union type at any time.

union member at most one stored 1426

C++

The C++ Standard does include union types within the definition of aggregate types, 8.5.1p1. So, this rationale was not thought applicable by the C++ Committee.

Other Languages

type category 553

Union types, as a type category, are unique to C (and C++), so this issue does not occur in other languages.

---

footnote 38

38) See 6.7.3 regarding qualified array and function types. 564

---

footnote 39

39) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions. 565

Commentary

footnote 31 509

The text of this footnote is identical to footnote 31; however, the rationale behind it is different. Type qualifiers did not exist prior to C90. Supporting a degree of interchangeability allows developers to gradually introduce type qualifiers into their existing source code without having to modify everything at once. Also source code containing old-style function declarations continues to exist. There is the possibility of pointers to qualified types being passed as arguments to such functions.

Other Languages

Most languages that contain type qualifiers, or modifiers, do not get involved in this level of implementation detail.

Coding Guidelines

Qualified and qualified version of the same type may appear as members of unions. This interchangeability of members almost seems to invite the side-stepping of the qualifier.

---

566 EXAMPLE 1 The type designated as "`float *`" has type "pointer to `float`". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type— its type is "pointer to const-qualified `float`" and is a pointer to a qualified type.

---

567 EXAMPLE 2 The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float`. Its type category is array.

---

568 **Forward references:** compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

---

569 The representations of all types are unspecified except as stated in this subclause.

types representation

Commentary

These representations are requirements on the implementation. Previous clauses in the C Standard specify cases where some integer types, complex types, qualified types and pointer types must share the same representation. Subclause 5.2.4.2.2 describes one possible representation of floating-point numbers. Other clauses in the standard specify when two or more types have the same representation. However, they say nothing about what the representation might be.

509 footnote 31
506 complex component representation
556 qualifiers representation and alignment
558 pointer to void same representation and alignment as
330 floating types characteristics

C90

This subclause is new in C99, although some of the specifications it contains were also in the C90 Standard.

C++

[Note: 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. 3.9p1

These C++ subclauses go into some of the details specified in this C subclause.

Other Languages

Most languages say nothing about the representation of types. The general acceptance of the IEC 60559 Standard means there is sometimes some discussion about using a floating-point representation that conforms to this standard. Java is intended to be processor-independent. To achieve this aim, the representation from a developer's point of view of all arithmetic types are fully specified.

Coding Guidelines

Code that makes use of representation information leaves itself open to several possible additional costs:

- The representation can vary between implementations. The potential for differences in representations between implementations increases the likelihood that there will be unexpected effort required to port programs to new environments.

- The need for readers to consider representation information increase the cognitive effort needed to comprehend code. This increase in required effort can increase the time needed by developers to complete source code related tasks.
- The failure to consider representation information by developers can lead to faults being introduced when existing code is modified.
- The additional complexity introduced by the need to consider representation information increases the probability that the maximum capacity of a reader's cognitive ability will be exceeded (i.e., the code will be too complicated to comprehend). Human response to information overload is often to ignore some of the information, which in turn can lead to an increase in the number of mistakes made.

cognitive effort 0

overconfidence 0

Developers often have a misplaced belief in their ability to use representation information to create more efficient programs (e.g., less time to execute or less machine code generated).

Cg 569.1 A program shall not make use of information on the representation of a type.

Dev 569.1 A program may make use of representation information provided this usage is documented and a rationale given for why it needs to be used.

---

object contiguous sequence of bytes

Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. 570

Commentary

A bit-field can occupy part of one or two bytes (it may occupy more than two bytes, but the third and subsequent bytes will be fully occupied). It can also share a byte with other bit-fields.

The representation of a type must contain sufficient bytes so that it can represent the range of values the standard specifies it must be able to represent. An implementation may choose to use more bytes; for instance, to enable it to represent a greater range of values, or to conform to some processor storage requirement. Requiring that the bytes of an object be contiguous only becomes important when the object is operated on as a sequence of bytes. Such operations are part of existing practice; for instance, using the library function `memcpy` to copy arrays and structures. The number of bytes in an object can be found by using the **`sizeof`** operator.

sizeof result of 1119

In the case of pointers, the response to DR #042 (which involved accesses using `memcpy`, but the response has since been taken to apply to all kinds of access) pointed out that " . . . objects are not "the largest objects into which the arguments can be construed as pointing."", and " . . . a contiguous sequence of elements within an array can be regarded as an object in its own right." and "the non-overlapping halves of array . . . can be regarded as objects in their own rights." In the following example the storage `p_1` and `p_2` can be considered to be pointing at different objects:

```
1  #include <string.h>
2
3  #define N 20
4
5  char a[2*N];
6
7  void f(void)
8  {
9  char *p_1 = a,
10       *p_2 = a+N;
11
12 /* ... */
13 memcpy(p_2, p_1, N);
14 }
```

The ordering of bytes within an object containing more than one of them is not specified, and there is no way for a strictly conforming program can obtain this information. The order that bytes appear in may depend on how they are accessed (e.g., shifting versus access via a pointer).

The terms *little-endian* (byte with lowest address occupies the least significant position) and *big-endian* (byte with lowest address occupies the most significant position) originate from Jonathan Swift's book, *Gulliver's Travels*. Swift invented the terms to describe the egg-eating habits of two groups of people who got so worked up about which way was best that they went to war with each other over the issue. endian

C++

An object of POD[4] type (3.9) shall occupy contiguous bytes of storage. 1.8p5

The acronym POD stands for *Plain Old Data* and is intended as a reference to the simple, C model, or laying out objects. A POD type is any scalar type and some, C compatible, structure and union types.

In general the C++ Standard says nothing about the number, order, or encoding of the bytes making up what C calls an object, although C++ does specify the same requirements as C on the layout of members of a structure or union type that is considered to be a POD.

Other Languages

Java requires integer types to behave as if they were contiguous; how the underlying processor actually represents them is not visible to a program.

Common Implementations

The Motorola DSP563CCC[967] uses two 24-bit storage units to represent floating-point values. The least significant 24 bits is used to represent the exponent (in the most significant 14 bits, the remaining bits being reserved). The significand is represented in the most significant storage unit (in two's complement).

Table 570.1: Byte order (indicated by the value of the digits) used by various processors for some integer and floating types, in different processor address spaces (all address spaces if none is specified).

Vendor	16-bit integer	32-bit integer	64-bit integer	32-bit float	64-bit float
AT&T 3B2		4321 (data space)/ 1234 (program space)			
DEC PDP–11	12	3412		3412 (F format)	78563412 (D format)
DEC VAX	12	1234	12345678	3412 (F format)	78563412 (D format)
NSC32016		1234 (data space)/ 4321 (program space)			

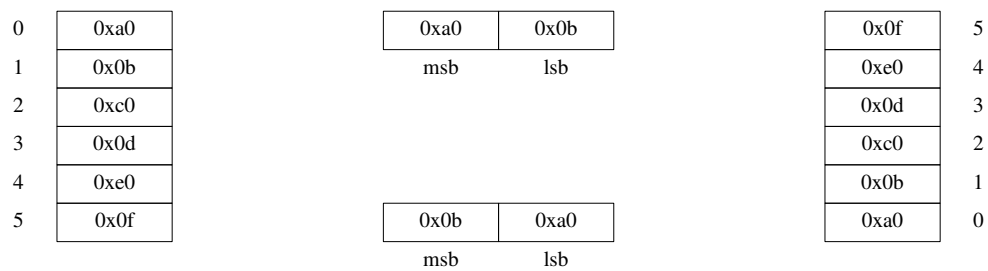


Figure 570.1: Developers who use little-endian often represent increasing storage locations going down the page. Developers who use big-endian often represent increasing storage locations going up the page. The value returned by an access to storage location *0*, using a pointer type that causes 16 bits to be read, will depend on the *endianness* of the processor.

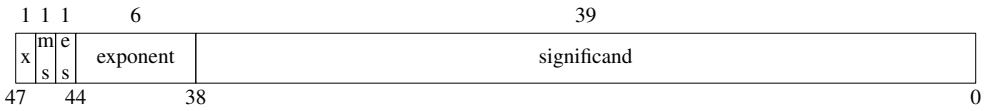


Figure 570.2: The Unisys A Series[1389] uses the same representation for integer and floating-point types. For integer values bit 47 is unused, bit 46 represents the sign of the significand, bits 45 through 39 are zero, and bits 38 through 0 denote the value (a sign and magnitude representation). For floating values bit 47 represents the sign of the exponent and bits 46 through 39 represent the exponent (the representation for double-precision uses an additional word with bits 47 through 39 representing higher order-bits of the exponent and bits 38 through 0 representing the fractional portion of the significand).

Coding Guidelines

Information on the number of bytes in a type is needed by the memory-allocation functions. The **sizeof** operator provides a portable way of obtaining this information. It is common C practice to copy values from one object to another using some form of block copy of bytes between storage locations. Perhaps part of the reason for this is lack of awareness, by developers, that objects having a structure type can be assigned, or in the case of objects having an array type, because there is no appropriate assignment operator available for this type category.

sizeof 1118 constraints

There are no portable constructs that provide information on the order or encoding of the bytes in an object. The only way to obtain this information is to use constructs whose behavior is implementation-defined. For these cases the guideline recommendation on using representation information is applicable.

representation information using 569.1

---

unsigned char pure binary

Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[40)] 571

Commentary

This is a requirement on the implementation. It prevents an implementation from using any of the bits in one of these types for other purposes. For instance, an implementation cannot define the type **unsigned char** to be 9 bits, the most significant bit being a parity bit and the other bits being the value bits; all 9 bits would have to participate in the representation of the type (or the byte size reduced to 8 bits).

This requirement implies that using the type **unsigned char** to access the bytes in an object guarantees that all of the bits in that object will be accessed (read or written). There is no such requirement for any other types. For instance, the type **int** may contain bits in its object representation that do not participate in the value representation of that object. Taking the address of an object and casting it to pointer-to **unsigned char** makes all of the bits in its object representation accessible. This requirement is needed to implement the library function `memcpy`, among others.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

3.9.1p1 *For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types.*

The C++ Standard does not include unsigned bit-fields in the above requirement, as C does. However, it is likely that implementations will follow the C requirement.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

representation information using 569.1

The lack of such a guarantee for the other types only becomes visible when programs get involved with details of the representation. The guideline on not using representation information is applicable here.

572 Values stored in non-bit-field objects of any other object type consist of *n* × **CHAR_BIT** bits, where *n* is the size of an object of that type, in bytes.

Commentary

This means that non-bit-field objects cannot share the sequence of bytes they occupy with any other object (members of union types share the same storage, but only one value can be stored in such a type at any time).

C90

This level of detail was not specified in the C90 Standard (and neither were any of the other details in this paragraph).

C++

The object representation of an object of type T is the sequence of N ***unsigned char*** *objects taken up by the object of type T, where N equals* `sizeof(T)`. 3.9p4

That describes the object representation. But what about the value representation?

The value representation of an object is the set of bits that hold the value of type T. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.[37)] 3.9p4

This does not tie things down as tightly as the C wording. In fact later on we have:

For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. 3.9.1p1

QED.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

Developers sometimes calculate the range of values that a type, in a particular implementation, can represent based on the number of bits in its object representation (which for most commonly used processors will deliver the correct answer). Such calculations are the result of developers focusing too narrowly on the details in front of them. There are object-like macros in the **<limits.h>** header that provide this information.

```
#include <limits.h>

unsigned short lots_of_assumptions(void)
{
return (1 << (sizeof(short)*CHAR_BIT)) - 1;
}
```

The following is a special case of the guideline recommendation dealing with the use of representation information (just in case developers regard this as a special case). 569.1 representation information using

Rev 572.1 The possible range of values that an integer type can represent shall not be calculated from the number of bits in its object representation.

value copied using unsigned char

The value may be copied into an object of type **unsigned char [***n***]** (e.g., by **memcpy**); 573

Commentary

Using the type **unsigned char** guarantees that all of the bits in the original object representation will be copied because there cannot be any unused bits in that type. While code may be written to copy the value into an array of any type, the standard only guarantees the behavior for this case, and when the element type is the same as the object being copied. Once the value has been copied into an object having type array of **unsigned char** copying them back to an object of the original type will restore the original pattern of bits, and hence the original value. The type pointer to **unsigned char** is a commonly used interface mechanism that offers a generic way of copying one region of storage to another region of storage. Although the library functions provide this functionality, there is existing code that does such copies as loops within the code itself rather than via calls to library functions.

C90

This observation was first made by the response to DR #069.

Other Languages

Many languages do not support the equivalent of the memory copying and related library functions. Neither do they define the behavior of casting pointers to different types, which enables developers to provide such functionality. So this discussion does not apply to them.

Common Implementations

All implementations known to your author use the same number of bits in the representation of the types **unsigned char** and **signed char** and there are no trap representations in the object representation of the type **signed char**. In such implementations values may be copied using an array of any character type.

Coding Guidelines

Copying objects a byte at a time is making use of representation information. However, this usage is sufficiently common in existing source to be regarded as a cultural norm.

Dev 569.1 An object may be copied by treating its contents as a sequence of objects having type **unsigned char**.

object representation

the resulting set of bytes is called the *object representation* of the value. 574

Commentary

value representation 595

This defines the term *object representation* (it was first used by the response to DR #069). An object representation is the complete sequence of bits making up any object. It differs from the value representation in that it may contain additional bits (which do not form part of the value). An object representation may be thought of in terms of being a sequence of bits, while the value representation is the interpretation of the contents of storage according to a type.

Other Languages

The term *object* usually has a very different meaning in object-oriented languages. In other languages, this term, if used, is often not defined in this way.

Coding Guidelines

The term *object representation* was introduced in C99 and is not used by developers in this context. Many developers have some familiarity with object-oriented languages. It is likely that they will assume this term has a very different meaning from its actual C99 definition. It is probably a good idea not to use the term *object representation* in guidelines unless plenty of explanatory information is also provided on the terminology.

575 Values stored in bit-fields consist of *m* bits, where *m* is the size specified for the bit-field.

bit-field value is m bits

Commentary

This is a requirement on the implementation. Bit-fields cannot contain any padding bits. A previous requirement specified the representation for bit-fields having an unsigned type. The integer constant given in the declaration of the bit-field member corresponds to the value of *m*. There is a maximum value that *m* can take.

571 unsigned char pure binary

1393 bit-field maximum width

C++

> *The* `constant-expression` *may be larger than the number of bits in the object representation (3.9) of the bit-field's type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field.*

9.6p1

This specifies the object representation of bit-fields. The C++ Standard does not say anything about the representation of values stored in bit-fields.

C++ allows bit-fields to contain padding bits. When porting software to a C++ translator, where the type **`int`** has a smaller width (e.g., 16 bits), there is the possibility that some of the bits will be treated as padding bits on the new host. In:

```
1  struct T {
2             unsigned int m1:18;
3            };
```

the member `m1` will have 18 value bits when the type **`unsigned int`** has a precision of 32, but only 16 value bits when **`unsigned int`** has a precision of 16.

Other Languages

A few languages (e.g., Ada, CHILL, and PL/1) provide functionality for enabling developers to specify the number of bits of storage to use in representing objects having integer types. Languages in the Pascal family support the specification of types that are subranges of the integer type. Some implementations chose to pack members of records having such types into the smallest number of bits, effectively having the same outcome as a C bit-field definition. But this implementation detail is hidden from the developer and an implementation is at liberty to use exactly the same amount of storage for all integer types.

576 The object representation is the set of *m* bits the bit-field comprises in the addressable storage unit holding it.

Commentary

This defines the object representation for bit-fields. It differs from the object representation for other types in that it is based on addressable storage units rather than bytes. This is because the storage occupied by a bit-field might only partially fill two bytes and share that storage with other bit-fields.

53 byte addressable unit

577 Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

Commentary

To be exact, two object representations whose values are interpreted using the same type can compare equal (using an equality operator) and have different object representations (two object representations can only be compared by treating them as an array of **`unsigned char`** objects and can be compared by using, for instance, the library function `memcmp`). This difference can occur if there are padding bits in the object representation that differ. Thus, although the values compare equal, the object representations are different.

1212 equality operators syntax

Two object representations can have the same bit pattern, but compare unequal when interpreted using two different types (e.g., integer and floating types).

There are also cases that do not involve padding bits in the object representation, two values comparing equal, but the object representations being different. This can occur when there is more than one representation for the same value; for instance, plus and minus zero in signed magnitude notation, or when the values are different but compare equal (i.e., plus and minus floating-point zero).

x == y 604
x not same as y

NaN 339

NaN always compares unequal to anything, including NaN.

C++

3.9p3 *For any POD type T, if two pointers to T point to distinct T objects `obj1` and `obj2`, if the value of `obj1` is copied into `obj2`, using the `memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`.*

This handles the first case above. The C++ Standard says nothing about the second value compare case.

Coding Guidelines

This statement highlights the dangers of dealing with object representations. They can contain padding bits whose value may not be explicitly controlled by developers.

Example

```
1  #include <string.h>
2
3  extern int glob_1,
4             glob_2;
5
6  int f(void)
7  {
8  if (memcmp(&glob_1, &glob_2, sizeof(glob_1)) == 0)
9     {
10    if (glob_1 == glob_2)
11       return 1;
12    else
13       return 2;
14    }
15 return 3;
16 }
```

Certain object representations need not represent a value of the object type. 578

Commentary

On some processors, pointers have a limit on the maximum amount of memory they can access; for instance, 24 bits of a possible 32-bit pointer representation can refer to an object.

The IEC 60559 Standard specifies that certain bit patterns do not represent floating-point numbers. However, these bit patterns are still values within the standard. They represent such quantities as the infinities and NaNs. These are all values of the object type.

floating types 338
can represent

C90

This observation was not explicitly made in the C90 Standard.

C++

3.9p4

The value representation of an object is the set of bits that hold the value of type T. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.[37)]

37) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Languages C.

Footnote 37

By implication this is saying what C says. It also explicitly specifies that these representation issues are implementation-defined.

Other Languages

Most languages do not get involved in specifying this level of detail.

Common Implementations

This situation is sometimes seen for pointer types. A processor may place restrictions on what storage locations can be addressed (perhaps because the program does not have access rights to them, or because there is no physical storage at those locations). The representation of the floating-point significand on the Motorola DSP563CCC[967] reserves the bit patterns `0x000001` through `0x3fffff` and `0xc00000` through `0xffffff`.

Coding Guidelines

Although such object representations may exist, creating a value having such a representation invariably relies on undefined or implementation-defined behavior:

- An object that has not been initialized can contain an object representation that is not a value in the object type.
- For pointer and floating-point types, there are implementations where some object representations do not represent a value of the object type. Creating such a representation invariably requires modifying the object through types other than the underlying pointer or floating-point type.

---

579 If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined.

trap representation reading is undefined behavior

Commentary

The pattern of bits held in an object has no meaning until they are interpreted as having a particular type. A pattern of bits that does not correspond to a value of the type used for the read access may be treated by the host processor in special ways— for instance, raising an exception. This statement points out this fact.

The exception for character types is to support the practice of copying objects using a pointer-to character type. This would not work if accessing one or more values, having a character type, represented undefined behavior.

C90

The C90 Standard specified that reading an uninitialized object was undefined behavior. But, it did not specify undefined behaviors for any other representations.

C++

The C++ Standard does not explicitly specify any such behavior.

Other Languages

Most languages specified some form of undefined or implementation-defined behavior for a read access to an uninitialized object. But they are usually silent on other kinds of *non-values*.

Common Implementations

There are so few processors that have a sequence of bits in the value representation that does not represent a value, it is not possible to determine any common behaviors.

pointer 454 cause undefined behavior
pointer 590 segmented architecture

Some processors check that the value of address operands is within the bounds of addressable storage. This kind of checking is common in processors that represent addresses using a *segment+offset* representation. In this case the segment number is often checked to ensure that it denotes a segment that can be accessed by the current process.

Coding Guidelines

object 461 initial value indeterminate representation 569.1 information using

A guideline recommending that such values not be read is acting after the fact, they should not have been created in the first place. These issues are dealt with under uninitialized objects and making use of representation information.

If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[41)] 580

Commentary

The representation could be produced by doing one of the following:

- Assigning it from another object of the same type; in which case the behavior is undefined because of the previous sentence.
- Treating all or part of the object as a type different from its declared type; in which case the behavior is also undefined.

effective type 948

- manipulating the values of bits directly using a sequence of operations on operands. While creating the value may not be undefined behavior, storing it into the appropriate type could be.

If a pointer to character type is used to copy all of the bits in an object to another object, the transfer will be performed a byte at a time. A trap representation might be produced after some bytes have been copied, but not all of them. An exception to the general case is thus needed for character types.

C90

The C90 Standard did not explicitly specify this behavior.

C++

The C++ Standard does not explicitly discuss this issue.

Other Languages

Other languages do not usually give special status to values copied using character types.

Common Implementations

Few implementations do anything other than treat whatever bit pattern they are presented with as a value. In other cases processors raise some kind of exception, or set some status bits.

Coding Guidelines

object 960 value accessed if type

These side effects are dealt with in more detail elsewhere.

---

trap representation

Such a representation is called a *trap representation*. 581

Commentary

This defines the term *trap representation*. The standard does not require that use of this representation cause a processor to generate trap (the original source for this terminology). As specified in the previous two sentences, the behavior is undefined. The term *trap representation* is not often used by developers because there are few implementations that contain such a representation. The Committee's response to DR #222 said, in part: "A TC should remove the notion of objects of struct or union type having a trap representation . . . ".

C90

This term was not defined in the C90 Standard.

C++

Trap representations in C++ only apply to floating-point types.

Common Implementations

Support for a trap representations outside of pointer and floating-point types is very rarely seen. A few processors use tagged memory; for instance, the Tera computer system[24] has four access bits for each 64-bit object, two of these bits are available for the translator implementor to use. Possible uses include stack limit checking and runtime type exception signaling. (If a location's trap bit is set and the corresponding trap-disable bit in the pointer is clear, a trap will occur.)

The IEC 60559 Standard specifies support for signaling NaNs. While these trigger the invalid exception when accessed, they are not trap representations (although their use may be intended to act as a trap for when certain kinds of situations occur during program execution).

Note. Some processors are said to *trap* for reasons other than accessing a trap representation. For instance, use of an invalid instruction is often defined, by processor specifications, to cause a trap.

---

582 40) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position.

footnote 40

Commentary

The bit with the highest position might be used as a sign bit.

C90

Integer type representation issues were discussed in DR #069.

Other Languages

Most languages rely on the definitions given for this representation provided by other standards and don't repeat them in modified form.

---

583 (Adapted from the *American National Dictionary for Information Processing Systems*.)

Commentary

The ISO definition is in ISO 2382–1.

25 ISO 2382

---

584 A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\mathbf{CHAR\_BIT}} - 1$.

unsigned char value range

Commentary

This duplicates requirements given elsewhere for CHAR_BIT and UCHAR_MAX.

307 CHAR_BIT macro
328 UCHAR_MAX value

C++

The C++ Standard includes the C library by reference, so a definition of CHAR_BIT will be available in C++.

> *Unsigned integers, declared* ***unsigned****, shall obey the laws of arithmetic modulo* 2^n *where* n *is the number of bits in the value representation of that particular size of integer.*[41)]

3.9.1p4

From which it can be deduced that the C requirement holds true in C++.

---

585 41) Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

footnote 41

Commentary

Such initialization could be performed, for instance, by the implementation on function entry. However, an implementation does not have access to any special instructions that could not also be used in the translation of the rest of a C program. It would need to make use of the exception granted for character types. An implementation could also choose to not explicitly initialize such a variable (the most common situation), using the pattern of bits it happens to contain. Whether this pattern of bits is a trap representation is something that an implementation does not need to be concerned about.

object 461 initial value indeterminate

C90

The C90 Standard did not discuss trap representation and this possibility was not discussed.

C++

The C++ Standard does not make this observation about possible implementation behavior.

Other Languages

This technique, for helping developers find uninitialized variables, has been used by implementations of various languages.

Common Implementations

The technique of giving uninitialized objects a representation that caused some form of unexpected behavior, if they are read without first being given an explicit value in the code, has long been used by implementations. It predates the design of the C language.

---

value stored in structure
value stored in union

When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[42)] 586

Commentary

footnote 601 42

As the footnote points out, there are two possible ways of copying structure or union objects. Depending on the method used any padding bits may, or may not, also be copied.

Scalar types may include padding bits in their representation, structure types may have padding between members, and both structure and union types may have padding after the last member.

structure 1424 unnamed padding
structure 1428 trailing padding
footnote 1042 DR283

The response to DR #283 clarified the situation with regard to wording that appeared in C90 but not C99.

C90

The sentences:

> *With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined 41).*

> *41) The "byte orders" for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately sized array of character type), but must be accounted for when conforming to externally-imposed storage layouts.*

appeared in C90, but does not appear in C99.

If a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined in C90 and unspecified in C99.

C++

This specification was added in C99 and is not explicitly specified in the C++ Standard.

Other Languages

The values of padding bytes, or even the very existence of such bytes, is not usually discussed in other languages.

Common Implementations

Many implementations simply copy all of the bytes making up an object having structure or union type; it is the simplest option. Whether the copy occurs a byte at a time or in multiple bytes (it is usually much more efficient to copy as many bytes as possible in a single instruction) is an implementation detail that is hidden from the developer. For padding bytes to play a part in the choice of algorithm used to make the copy there would have to be a significant percentage of the number of bytes needing to be copied.

In the case of an object having union type a translator may, or may not be able to deduce which member was last assigned to. In the former case it can copy the member assigned to, while in the latter case it has to assume that the largest member is the one that has to be copied. Whether padding bytes are copied can depend on the context in which the copy occurs.

Cyclone C[668] supports tagged union types, which contain information that identifies the current member.

Coding Guidelines

A program cannot rely on the padding bits of an object having structure or union type remaining unchanged when one of its members is modified, or the entire object is assigned a new value. Similarly, a program cannot assume that because one structure object was assigned to another structure object, a call to `memcmp` will return zero.

Copying a structure object using the assignment operator is still not widely used by developers, even though support for both structure assignment, parameter passing, and functions returning a structure has been available since the C90 Standard. Many developers continue to use the `memcpy` library function to copy objects having structure type. In this case any padding bytes will also be copied and the two objects will compare equal after the copy operation.

The issue of copying and assigning structure objects is discussed elsewhere. 601 footnote 42

Example

```
#include <string.h>

struct T {
          double m_1;
          short m_2;
         } x, y;

_Bool f(void)
{
x = y;
return memcmp(&x, &y, sizeof(x)) == 0;
}

void g(void)
{
union {
       char a[5];
       char b[2];
      } u;

u.a[4] = 7;
/*
 * The following assignment to the member b effectively turns
 * most of the elements of member a into padding bytes.
 */
u.b[1] = 2;
/*
 * After the following assignment it is not guaranteed
 * that u.a[4] has the value 7.
 */
```

```
31  u.a[0] = 3;
32  }
```

~~The values of padding bytes shall not affect whether the value of such an object is a trap representation.~~ The value of a structure or union object is never a trap representation, even though the value of a member of a structure or union object may be a trap representation. 587

Commentary

This is a requirement on the implementation. If it is possible for such a type to have a trap representation, it is the implementation's responsibility to ensure that the padding bytes never take on values that generate such a representation.

The wording was changed by the response to DR #222.

C90

This requirement is new in C99.

~~C++~~

This wording was added in C99 and is not explicitly specified in the ~~C++~~ Standard.

Common Implementations

Trap representations are not usually associated with non-scalar types.

~~Those bits of a structure or union object that are in the same byte as a bit-field member, but are not part of that member, shall similarly not affect whether the value of such an object is a trap representation.~~ 588

Commentary

This sentence was deleted by the response to DR #222.

As an example, consider an implementation where a particular pattern of bits within a 32-bit storage unit constitutes a trap representation. If a member of a structure type is declared to have a width of 28 bits and the translator assigns storage for it within the 32-bit storage unit without allocating any member to the remaining 4 bits, then it is the implementation's responsibility to ensure that either:

- These padding bits never take on values that generate a trap representation. This could enable the translator to generate machine code that loaded the value held in the 32-bit storage unit in a single instruction, followed by instructions to scrap off the unwanted 4 bits.
- For accesses to the bit-field, generate machine code that never loads the 32-bit storage unit directly, but uses a sequence of loads of narrower quantities (16 or 8 bits). These narrower quantities are joined together to create the value of the 28-bit wide bit-field. This ensures that, although the 32-bit storage unit may hold a trap representation it is never accessed as such.

Common Implementations

While a few processors[968] contain instructions that can load a specific number of bits from storage, it is likely that the underlying hardware operations will access the complete byte or word to extract the required bits. The number of processors that support bit-field access is low. Your author does not know of any processor supporting such instructions and a trap representation.

union member when written to

When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values~~, but the value of the union object shall not thereby become a trap representation~~. 589

Commentary

This is a requirement on the implementation. If trap representations are supported, then any padding bytes must never take on values that would cause the value of the union object to become a trap representation.

Any bytes in a union type's object representation that are not part of the representation of any of the members are padding bytes. A consequence of these bytes taking unspecified values is that accessing the value of other members is undefined behavior. (The standard does not require the value representation of any scalar type to completely overlap the value representation of another scalar type; their object representations may contain padding bytes at various offsets such that a complete overlap cannot occur.)

1428 structure trailing padding

The wording was changed by the response to DR #222.

Common Implementations

For non-bit-field members, most implementations store operation into a member object does not affect the values held in other bytes of a union object. For bit-field members, an implementation may chose to write a value into all of the bytes occupied by the storage unit containing the bit-field. (This is an optimization that simplifies the machine code generated for a read access; no masking operation is needed to zero bits that are not part of the value representation if store operations set them to zero.)

Trap representations, when they are supported, are invariably associated with scalar types only.

---

590 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[43)]

Commentary

This is a requirement on the implementation. Operators operate on the value representation, not the object representation. Padding bits, which are one way a value can have more than one object representation, do not affect operations on the value representation. It is also possible for there to be more than value representation denoting the same value.

602 footnote 43

C90

This requirement was not explicitly specified in the C90 Standard.

C++

This requirement was added in C99 and is not explicitly specified in the C++ Standard (although the last sentence of 3.9p4 might be interpreted to imply this behavior).

Other Languages

Most languages do not get involved in specifying this level of representation detail.

Common Implementations

While processors that have a linear address space create few complications for vendors of translators, there are a number of reasons why hardware vendors might want to divide up a processor's address space into discrete *segments* (as always the driving forces are cost and performance). The most well-known segmented architecture is the Intel x86 processor family (whose early family members' 64 K segment size was considered generous for the first few years of their life). Segmented architectures did not die with the introduction of the Pentium. Some vendors of 32-bit processors have introduced 2^{32} segment sizes as a way of transitioning existing code to 64-bit address spaces. At the other end of the scale, low-cost embedded processors may have a flat address space, but support pointers capable of accessing only part of it (instructions involving such short-range pointers invariably occupy less storage and are often faster than other pointers).

pointer segmented architecture

The following discussion is based on techniques adopted by vendors of translators targeting the Intel 80286 member of the Intel x86 family and supporting pointers that handle various forms of segmented addressing. Some, or all, of these issues are likely to be applicable to other processor architectures. There are several ways instructions can specify an address in storage on the Intel 80286, including the following:

- *Specifying a 16-bit offset value.* This offset value is added to the value held in a segment register (the choice of segment register, one of four, is implicit in the instruction) to form an address in storage.

This form is the simplest and fastest, but addressable storage was limited to a single 64 K segment. The value of a pointer object is the 16-bit offset (i.e., it is a 16-bit pointer).

- *Specifying a 16-bit pointer value and the segment register to use in forming the final address in storage.* This form does not have the 64 K restriction, but incurs the cost of having to generate code to load the chosen segment register with an appropriate value. The value of a pointer object has two parts — the 16-bit offset and the 16-bit value loaded into the segment register (i.e., it is a 32-bit pointer).

Pointers using the 16-bit representation are sometimes referred to as *near* pointers (after the keyword that is usually used in their declaration— e.g., `char near *p`); while pointers using the 32-bit representation are sometimes referred to as *far* pointers (after the keyword that is usually used in their declaration— e.g., `char far *p`).

Additional complications are caused by the fact that the two 16-bit components do not specify the value of a 32-bit address but the value in the 20-bit address space supported by the Intel 80286. The address is formed by left shifting the segment value by four and adding the offset to it (e.g., `(seg << 4) + offset`). This means that two different pairs of 16-bit values can specify the same address (e.g., the hexadecimal pairs *ABCD:0000* and *ABCC:0010* both specify the address *0xABCD0*). Additional complications arise because of vendors' desire to optimize the machine code generated for operations involving pointers. For instance, the machine code needed to compare or subtract two pointer values will depend on whether the operands are both near pointers (only need to deal with the offset on the basis that the segment register values will be the same) or some other combination (where both components need to be considered).

Using a normalized form for the representation of far pointers, where the bits from the two components don't overlap (e.g., arranging that the offset is always less than 0x10), allows equality and relational tests to be performed without having to add the two values together. However, the results obtained from pointer arithmetic have to be converted to a normalized form.

Developers writing programs that need to simultaneously contain more than 64 K of object data had to be very careful how they mixed any operations that involved the two kinds of pointers (the above discussion has been simplified and most implementations supported more than two kinds of pointers).

null pointer 750
conversion yields
null pointer

Some of the issues involved in the representation of the null pointer constant on a segmented architecture are discussed elsewhere.

---

object representation more than one

Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated. 591

Commentary

This is a requirement on the implementation. It ensures that a subsequent read operation can always access the value of the object without worrying that a previous store had stored a trap representation into it. There are ways of causing an object to hold a trap representation, but they all involve making use of undefined behavior. The standard is silent on the case where there is more than one possible value representation for the stored value.

C90

The C90 Standard was silent on the topic of multiple representations of object types.

C++

This requirement is not explicitly specified in the C++ Standard.

Common Implementations

pointer 590
segmented
architecture

The following discusses the case where there is more than one value representation for the same value. On segmented architectures, an implementation may define some canonical representation for pointer types (this has the advantage of simplifying some operations— e.g., pointer compare). All values are converted to this form when they are stored into a pointer object. An implementation may not exhibit the expected behavior if it reads a pointer value that does not use this canonical representation (which might be created by manipulating the object representation of an object having a pointer type).

592 **Forward references:** declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1).

6.2.6.2 Integer types

593 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter).

unsigned integer types object representation

Commentary

Padding bits can be used for several reasons:

- A value representation may be stored in a larger object representation. This can occur if a processor only has a single load and store instruction, which always operate on a fixed number of bits.
- The host processor uses additional bits to indicate properties of the object representation; for instance, parity of the stored value, or tagged data, where the tag specifies type stored at that location.
- Representations are shared between different types (either to reduce processor transistor count, or for backward compatibility with existing code); for instance, integer types and floating-point types, or integer types and pointer types.

C90

Explicit calling out of a division of the bits in the representation of an unsigned integer representation is new in C99.

C++

Like C90 the grouping of bits into value and padding bits is not explicitly specified in the C++ Standard (3.9p2, also requires that the type **unsigned char** not have any padding bits).

Other Languages

Most languages do not get involved in specifying this level of representation detail.

Common Implementations

On some Cray processors the type **short** has 32 bits of precision but is held in 64 bits worth of storage.

54 word addressing

The Unisys A Series unsigned integer type contains a padding bit that is treated as a sign bit in the signed integer representation (see Figure 570.2).

Coding Guidelines

569.1 representation information using

Programs whose behavior depends on the value of padding bits make use of representation information. The fact that modern processors rarely contain padding bits in their integer types does not mean that future processors will continue this trend. The main developer assumption that fails in the presence of padding bits is that there are `CHAR_BIT*sizeof(integer_type)` bits in the value representation. The only reliable way of finding out the number of bits in the value representation is to look at the `MIN_*` and `MAX_*` macros defined in the header **<limit.h>**.

300 numerical limits

594 If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^{N}-1$ using a pure binary representation;

Commentary

This is a requirement on the implementation. Representations ruled out by this requirement include BCD for integer types (some processors, including the Intel x86 processor family, contain instructions for operating on this representation) and Gray codes (consecutive decimal values are represented by binary values that differ in the state of a single bit; there is no requirement that ones start in the least significant bit and percolate up, so there are many possible bit sequences that are Gray codes).

Table 594.1: Pattern of bits used to represent decimal numbers using various coding schemes.

Decimal	Binary	Gray code	111 biased	2–out-of–5
0	0000	0000	0111	00011
1	0001	0001	1000	00101
2	0010	0011	1001	00110
3	0011	0010	1010	01001
4	0100	0110	1011	01010
5	0101	0111	1100	01100
6	0110	0101	1101	10001
7	0111	0100	1110	10010
8	1000	1100	1111	10100
9	1001	1101		11000
10	1010	1111		
11	1011	1110		
12	1100	1010		
13	1101	1011		
14	1110	1001		
15	1111	1000		

shift-expression syntax 1181

This requirement means there has to be a sequential ordering of the bits used in the value representation. A sequential ordering of bits means that the shift operators cause a uniform pattern of bit movement. For instance, the following two functions are equivalent:

```
#include <limits.h>

/*
 * This code assumes that sizeof(int) == 2
 */

unsigned int obj_htons(unsigned int h)
{
/*
 * Relies on type unsigned char copying all bits.
 */
unsigned char *hp = (unsigned char *)&h;
unsigned int n;
unsigned char *np = (unsigned char *)&n;

np[0] = hp[1];
np[1] = hp[0];
return n;
}

unsigned int val_htons(unsigned int h)
{
#define MASK ((1U << CHAR_BIT) - 1)
return ((h & MASK) << CHAR_BIT) | ((h >> CHAR_BIT) & MASK);
}
```

value bits signed/unsigned 600

Requirements on the value bits of signed integer types are phrased in terms of the value of the corresponding bits in the corresponding unsigned integer type. There is no requirement that other non-integer scalar types be represented using a binary notation.

C90

These properties of unsigned integer types were not explicitly specified in the C90 Standard.

Other Languages

Even those languages that contain an unsigned integer type do not specify this level of detail.

Coding Guidelines

When targeting hosts with relatively low-performance processors, there are advantages to making use of the mathematical properties of a pure binary representation— for instance, using the shift operator to replace multiplies and divides of positive values by powers of two; or using the sequence add, shift and add for multiplication by 10. A number of arguments against this usage are often heard:

- Performing these kinds of optimizations based on representation details at the source code level creates a dependency on that representation. However, the standard specifies what the representation must be.
- It is claimed that an optimizing translator would perform the mapping to shift and add instructions, and that developers should leave this kind of optimization to the translator. However, not all optimizers live up to their name.
- Using such constructs makes the code more difficult to comprehend. Is an expression that uses a shift operator, for instance, harder to comprehend than one that uses a multiply or divide? It is really a question of familiarity. Developers who frequently encounter this usage learn to quickly recognize it and comprehend its purpose. This is not true for developers that rarely encounter this usage.

The general issue is one of developer overconfidence and comprehension. Overconfidence, or perhaps tunnel vision, by the original developer in believing that an optimization performed at any point in the source will impact the efficiency of the program. Subsequent readers are likely to require greater cognitive resources to comprehend the use of sequences of bit manipulation operations that are intended to mimic the effects of an arithmetic operator. These issues are discussed in more detail in the respective operator sentences.

One issue, which is rarely considered, is the impact of bit manipulation on optimizers and static analysis tools. These do not usually analyze bit manipulation operations in terms of their arithmetic effect. They tend to treat these operations as creating a sequence of bits of unknown value. As such, these operations reduce the information that optimizers and static analysis tools are able to deduce from the source.

---

595 this shall be known as the value representation.

value representation

Commentary

This defines the term *value representation* (it was first used in the response to DR #069). Given that, for most implementations the value representation and the object representation are bit-for-bit identical, it seems unlikely that this term will become commonly used by developers.

---

596 The values of any padding bits are unspecified.[44)]

unsigned integer padding bit values

622 integer padding bit values

Commentary

This is a special case of a more general one specified elsewhere. A strictly conforming program cannot access the value of these bits. Given that programs are not intended to access the values of these padding bits, there is no obvious reason for specifying the value as being implementation-defined. Such a specification would have required implementations to document their behavior in this area, which is likely to vary between objects in different contexts and be generally very difficult to specify for all cases.

C90

Padding bits were not discussed in the C90 Standard, although they existed in some C90 implementations.

C++

This specification of behavior was added in C99 and is not explicitly specified in the C++ Standard.

Common Implementations

Some hosts have one or more parity bits associated with each storage byte. The purpose is to provide confidence that storage has not been corrupted (to at least the additional degree of probability provided by the number of parity bits used) or even corrected hardware errors (if enough parity bits are available). These parity bits are normally handled by the storage hardware and are not visible at the software level. (It is likely that the processor knows nothing about how the bytes in storage are handled.)

representation information using 569.1

Coding Guidelines

The guideline recommendation dealing with the use of representation information applies to access to padding bits.

---

signed integer types object representation unsigned 593 integer types object representation IEC 60559 29

For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. 597

Commentary

The difference in the representation, from unsigned integer types, is that there is a sign bit. An alternative representation of signed numbers, not using a sign bit, is to use biased notation. This method of representation is used for the exponent in the IEC 60559 floating-point standard.

Other Languages

All known computer languages can represent signed integer types, so their implementations have some means of representing the sign. But language specifications do not usually go into this level of detail.

Common Implementations

This division of the object representation into three groups makes it a superset of the representations used by all known processors (many do not have padding bits). The Harris/6 computer represented the type **long** using two consecutive **int** types. This meant that the sign bit of one of the ints had to be ignored; it was treated as a padding bit. The value representation of the type **int** is 24 bits wide, and **long** had a value representation of 47 bits with one padding bit. The Unisys A Series uses a representation for its signed integer types that contain all three groups (see Figure 570.2). Also, the sign bit is not part of the value representation of its unsigned integer types.

There need not be any padding bits; 598

Commentary

The standard places no upper limit on the number of padding bits that may exist in an integer representation.

Common Implementations

Most implementations have no padding bits.

---

sign one bit

there shall be exactly one sign bit. 599

Commentary

sign bit 610 representation

Having specified that there is a sign bit, the standard needs to say something about it. The encoding of the sign bit limits the number of possible integer representations to three (when a binary representation is used).

C90

This requirement was not explicitly specified in the C90 Standard.

C++

3.9.1p7 *[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.]*

These three representations all have exactly one sign bit.

Other Languages

Cobol provides several different arithmetic types. One of these types requires a representation based on using the Ascii character set values for the digits and the sign. This sign representation not only occupies more than one bit, but where it occurs in the object representation can vary. (It can occupy a single byte that leads or trails the digits, or it can be encoded as part of the value representation of the leading or trailing digit.)

600 Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are *M* value bits in the signed type and *N* in the unsigned type, then $M \leq N$).

value bits signed/unsigned

Commentary

This is a requirement on the implementation. The value bits in the signed integer representation must represent the same power of two as the corresponding bit in the unsigned representation. From this we can deduce that, provided there are no padding bits in the object representation of the unsigned type, the sign bit is the most significant bit.

The relational condition on the number of bits in each value representation is specified in terms of ranges of values elsewhere.

495 positive signed integer type subrange of equivalent unsigned type

C90

This requirement is new in C99.

C++

> *The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.* 3.9.1p3

If the value representation is the same, the value bits will match up. What about the requirement on the number of bits?

> *. . . each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[40]; that is, each signed integer type has the same object representation as its corresponding unsigned integer type.* 3.9.1p3

Combining these requirements with the representations listed in 3.9.1p7, we can deduce that C++ has the same restrictions on the relative number of value bits in signed and unsigned types.

Other Languages

Most languages do not contain both signed and unsigned types. Those languages that do, do not get involved in this level of detail.

Common Implementations

In most implementations there is one more bit in the value representation of an unsigned type than in its corresponding signed type (the sign bit). The Unisys A Series (see Figure 570.2) is the only processor known to your author where the number of value bits in both the signed and unsigned types are the same.

601 42)Thus, for example, structure assignment ~~may be implemented element-at-a-time or via **memcpy**.~~ need not copy any padding bits.

footnote 42

Commentary

The wording was changed by the response to DR #222, which also contained the following Committee discussion:

> *It was observed that the point of the original footnote was primarily to illustrate one reason why padding bits might not be copied: because member-by-member assignment might be performed. But member-by-member assignment would imply that struct assignment could produce undefined behavior if a member of the struct had a value that was a trap representation. Instead of adding further text explaining that member values that were trap representations were not permitted to render assignment of a containing struct or union object undefined (e.g., if member-by-member copying were used), it was decided that the footnote should simply clarify the issue of padding bits directly.* DR #222

C90

The C90 Standard did not explicitly specify that padding bits need not be copied.

C++

Footnote 36 *36) By using, for example, the library functions (17.4.1.2) `memcpy` or `memmove`.*

The C++ Standard does not discuss details of structure object assignment for those constructs that are supported by C. However, it does discuss this issue (12.8p8) for copy constructors, a C++ construct.

Example

```
1  #include <stdio.h>
2
3  struct T_1 {
4              int mem_1;
5              unsigned : 4;
6              int mem_2: 2;
7              double mem_3;
8             };
9  struct T_2 {
10             int mem_1;
11             unsigned int mem_name: 4;
12             int mem_2: 2;
13             double mem_3;
14            };
15 union T_3 {
16            struct T_1 su; /* First named member is initialized. */
17            struct T_2 sn; /* Both structure types have a common initial sequence. */
18           } gu;
19 struct T_1 s;
20
21 int main(void)
22 {
23 union T_3 lu;
24
25 lu.su = s;
26
27 if (lu.sn.mem_name != 0)
28    print("Whether this string is output is unspecified\n");
29 if (gu.sn.mem_name != 0)
30    print("Whether this string is output is unspecified\n");
31 }
```

footnote 43 43) It is possible for objects **x** and **y** with the same effective type **T** to have the same value when they are accessed as objects of type **T**, but to have different values in other contexts. 602

Commentary

For this situation to occur, the two objects must contain different bit patterns. But, how can two objects containing different sequences of bits ever be considered to contain the same value?

effective type 948

- When there are padding bits in the object representation. They are padding bits in the sense of not being part of the value representation of the object's effective type. When the object is treated as, for instance, an array of **`unsigned char`**, the padding bits are included in its value.

- When two different sequences of bits are interpreted using some type to have the same value; for instance, plus and minus zero in signed magnitude or IEC 60559 floating-point. Also, in some implementations, two different sequences of bits can represent the same address in storage.

604 x == y
x not same as
590 pointer
segmented
architecture

C90

This observation was not pointed out in the C90 Standard.

C++

The C++ Standard does not make a general observation on this issue. However, it does suggest that such behavior might occur as a result of the **reinterpret_cast** operator (5.2.10p3).

Other Languages

Most languages do not get involved in specifying this level of detail.

Common Implementations

Very few implementations have padding bits in the object representation of arithmetic types. Most implementations use two's complement notation for integer types. Each value has a unique *value representation* in this notation.

In the past some processors have used 32 bits in the object representation of pointer types, but used fewer bits in their value representation. As the price of memory chips dropped and customers demanded support for greater amounts of storage capacity, vendors upgraded their processors to use the full 32 bits of representation. Processors using 64 bits in their pointer type object representation are now available, but it is expected that it will be some years before memory chip prices reach the point where this capability will be fully utilized. At the time of this writing some processors only use 48 bits effectively making the remaining 16 bits padding.

On segmented architectures, there is often more than one sequence of bits (pointer value representations) capable of denoting the same address.

590 pointer
segmented
architecture

Coding Guidelines

This footnote highlights the dangers of developers becoming involved in the representation details of objects.

569 types
representation

Example

```
1  _Bool negative_zero_may_have_more_than_one_representation(int valu)
2  {
3  if (valu == 0)
4     if (value & 1) /* Representation used affects the result here. */
5        return 1;
6  return 0;
7  }
```

---

603 In particular, if == is defined for type **T**, then **x == y** does not imply that **memcmp(&x, &y, sizeof (T)) == 0**.

Commentary

The equality operators are not defined for operands having structure or union types, so padding bytes are not an issue here. It is also true that

1213 equality
operators
constraints

- `x != y` does not imply `memcmp(&x, &y, sizeof(T)) != 0` (this case occurs if both `x` and `y` have the same NaN value);
- `(x-y) == 0` does not imply `x == y` (it need not apply for nonzero values of `x` and `y` if an implementation does not support subnormals);
- on a segmented architecture, there may be more than one representation of the null pointer constant; this is discussed elsewhere.

1164 subtraction
result of
338 subnormal
numbers

590 pointer
segmented
architecture

Other Languages

Very few languages define a function that is the equivalent of `memcmp`, so this is rarely an issue. However, some implementations of these languages provide an equivalent function, although such implementations do not usually go into the implications of its use.

Coding Guidelines

representation information using 569.1

Calling `memcmp` is making use of representation information, even though developers do not see it in that light. The guideline recommendation dealing with the use of representation information is applicable. Given that C does not contain support for objects having structure or union types as operands of the equality operators, use of `memcmp` has some attractions. Used in conjunction with the **`sizeof`** operator, it does not need modification when new members are added to the types. However, padding bytes do need to be taken into account.

---

x == y
x not same as y

Furthermore, **`x == y`** does not necessarily imply that **`x`** and **`y`** have the same value; 604

Commentary

equality operators true or false 1219

The values `-0.0` and `+0.0` are different values (dividing by these values results in -∞ and -∞, respectively), but they compare equal. If x and y have different integer types, there are several different cases where they can compare equal and have different values; for instance (assuming the type **`int`** is 16 bits and the type **`long`** is 32 bits), the values `-1` and `0xffff` compare equal.

pointers compare equal 1233

For pointer types, equality is defined in terms of them pointing to the same object.

Other Languages

The values that create this behavior in C are caused by the underlying representations used by the host processor. As such, the cases are applicable to all languages supporting the same C representations that are translated for execution on that processor.

equality operators syntax 1212

Some languages provide more than one mechanism for comparing for equality. This issue is discussed elsewhere.

Coding Guidelines

equality operators not floating-point operands 1214.1

Modern processors have a single representation for integer zero. If the guideline recommendation dealing with comparing objects having floating-point types for equality is followed, there will not be any issues for those types.

Example

```
1  #include <stdio.h>
2
3  extern double x, y;
4
5  void f(void)
6  {
7  if (x == y)
8     if ((1.0 / x) != (1.0 / y))
9        printf("x and y are differently signed zeros\n");
10 }
11
12 void g(void)
13 {
14 unsigned int x = 0xffff;
15 signed int y = -1;
16 unsigned long z = 0xffff;
17
18 if (x == y)
19    if ((x == z) == (y == z))
20       printf("int and long probably have the same representation\n");
21    else
```

```
22          printf("int and long probably have a different representation\n");
23      }
```

605 other operations on values of type `T` may distinguish between them.

Commentary

In the case of integer types represented using signed magnitude or one's complement notation, there are two representations for zero, −0 and +0. Arithmetic operations always deliver the same result. In the case of bitwise operations, their behavior on signed operands is undefined, so the issue of two representations is moot.

946 bitwise operations signed types

There are also two representations of zero, `-0.0` and `+0.0`, in IEC 60559. A great deal of thought lay behind the decision to have a signed zero and the results of arithmetic operations on them (in some cases the result depends on the sign, while in others it does not). For instance, dividing by −0 returns -∞ and dividing by +0 returns +∞ (they may produce the same results using other floating-point models), but adding a nonzero value always delivers the same result. The `copysign` library function will also return different results.

338 floating types can represent
342 signed of non-numeric values

Coding Guidelines

This situation usually arises when there is more than one representation of zero. This occurs for integer types represented in one's complement and signed magnitude format, or the floating-point representation in IEC 60559. The number of defined operations that can be performed using pointer types is sufficiently limited that multiple representations are not an issue.

In the case of IEC 60559 the decision to have signed zeros was intended to be an aid to the developer writing numerical software. If unexpected behavior occurs in this case, the cause is likely to be developer misunderstanding of the mathematics involved, which is outside the scope of these coding guidelines.

606 44) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit.

footnote 44

Commentary

The trap representations that occur because of parity errors are usually caused by hardware rather than software faults, while parity bits are not limited to those supported by hardware. It is rare for an implementation to choose to use one or more bits for this purpose.

This footnote is identical to footnote 45.

628 footnote 45

C90

This footnote is new in C99.

C++

This wording was added in C99 and is not explicitly specified in the C++ Standard. Trap representations in C++ only apply to floating-point types.

Common Implementations

Processors that contain a trap representation for integer types (apart from parity bits, which are not usually visible at the program level) are rare. At least one processor includes a parity bit in its floating-point representation.

335 WE DSP32

Coding Guidelines

Any program that accesses a subpart of an object to manipulate padding bits is making use of representation information and is covered by a guideline recommendation.

569.1 representation information using

607 Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types.

arithmetic operation exceptional condition

Commentary

divide by zero 1150

This statement excludes arithmetic operations, not bitwise operations. While divide by zero (or the remainder operation) often raises an exception, the standard specifies the result of such an operation as being undefined.

C++

This discussion on trap representations was added in C99 and is not explicitly specified in the C++ Standard.

Common Implementations

While various processor status flags might be set as a result of an arithmetic operation, they are not part of the representation of a value.

All other combinations of padding bits are alternative object representations of the value specified by the value bits. 608

Commentary

The object representation includes all bits in the storage allocated to the object, while the value representation may be a subset of these bits. The standard treats the combination of the value bits and the different values of the padding bits as alternative object representations.

C++

This discussion of padding bits was added in C99 and is not explicitly specified in the C++ Standard.

If the sign bit is zero, it shall not affect the resulting value. 609

Commentary

This is a requirement on the implementation. It reflects existing processor integer representation practice.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

3.9.1p7 *[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.]*

In these three representations a sign bit of zero does not affect the resulting value.

Other Languages

Other languages do not usually go into the representation details of their supported integer types. However, implementations of these languages will target the same hosts as C implementations. Unless a nonbinary representation is used to represent integer types, these implementations will follow the C model.

---

sign bit representation

If the sign bit is one, the value shall be modified in one of the following ways: 610

Commentary

This is a requirement on the implementation. These three ways correspond to the three representations of binary notation in common (well one of them is) usage.

C90

This requirement was not explicitly specified in the C90 Standard.

---

sign and magnitude

— the corresponding value with sign bit 0 is negated (*sign and magnitude*); 611

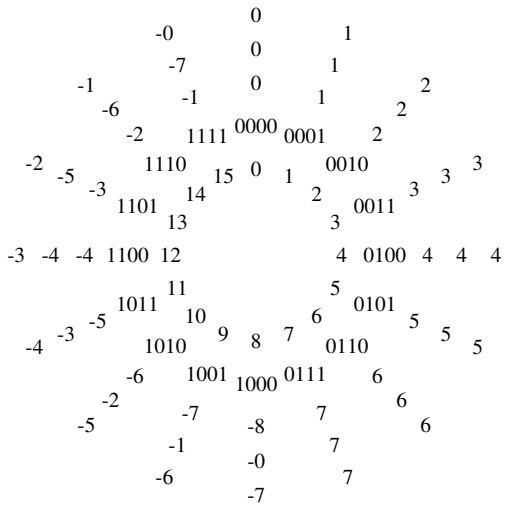


Figure 610.1: Decimal values obtained by interpreting a sequence of bits in various ways. From the inside out: unsigned, binary, two's complement, sign and magnitude, and one's complement.

Commentary

Sign and magnitude is a symmetric representation (i.e., the minimum representable value is the negated form of the maximum representable value). Every value, including zero, has the same representation for its magnitude (rather like putting +/– on the front of an integer constant). This representation is used for the significand in the IEC 60559 Standard.

317 int minimum value

C++

Support for sign and magnitude is called out in 3.9.1p7, but the representational issues are not discussed.

Common Implementations

Integer types, excluding the character types, on the Unisys A Series[1389] use sign and magnitude representation.

---

612 — the sign bit has the value -(2^N) (*two's complement*); two's complement

Commentary

When this value (i.e., the sign bit) is added to the value of the numeric quantity, the result is the bit pattern held in storage. The set of possible representable values are asymmetric about zero and there is one representation of zero.

C++

Support for two's complement is called out in 3.9.1p7, but the representational issues are not discussed.

Common Implementations

This is by far the most common representation in use today (and for the last 30 years). Even the translator for the Unisys A Series, whose underlying representation is sign and magnitude, provides an option to emulate two's complement.[1390]

---

613 — the sign bit has the value $-(2^N - 1)$ (*one's complement*). one's complement

Commentary

From the developers point of view, one's complement differs from two's complement in that the set of possible representable values are symmetric about zero and there are two representations of zero.

C++

Support for one's complement is called out in 3.9.1p7, but the representational issues are not discussed.

Common Implementations

The CDC 6600, 7600, and Cyber 200 are the only known (to your author) commercially sold processors (no longer available) using one's complement representation that supported a C translator.[852]

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for one's complement), is a trap representation or a normal value. 614

Commentary

Implementations with such trap representations are thought to have existed in the past. Your author was unable to locate any documents describing such processors.

C90

The choice of representation for signed integer types was not specified as implementation-defined in C90 (although annex G.3.5 claims otherwise). The C90 Standard said nothing about possible trap representations.

C++

The following suggests that the behavior is unspecified.

3.9.1p7 *The representations of integral types shall define values by use of a pure binary numeration system[44]. [Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.]*

Trap representations in C++ only apply to floating-point types.

Other Languages

Most languages do not get involved in specifying this level of detail.

Common Implementations

The Unisys A Series[1390] uses two's complement for character types and a sign and magnitude representation for the other integer types.

Coding Guidelines

The use of two's complement, by commercially available processors, is almost universal. Is there any worthwhile benefit in writing programs to execute correctly using all three representations?

Calculating the cost/benefit requires estimating the probability of having to port to a non-two's complement processor and the costs of writing code that has the same behavior for all integer representations likely to be used. There is only one commercially available processor (known to your author) that does not use two's complement, the Unisys A Series.[1389] The probability of having to port code to this platform can only be calculated by individual development groups. Because of the likelihood that the overwhelming volume of existing code contains some representation dependencies, there is every incentive for hardware vendors to continue to use two's complement in new processors.

There are two ways a program can depend on details of the representation— by explicitly manipulating the bits of the value representation using bitwise operators, and by generating values that are not guaranteed to be representable in all implementations of an integer type. Manipulating bits rather than numeric values is generally recommended against for reasons other than portability to other representations.

types 569 representation

Is simply specifying use of a two's complement representation sufficient? The minimum integer limits specified by the C Standard are symmetrical. However, all two's complement implementations (known to your author) are not symmetrical. They support one more negative value than positive value. Writing programs that only make use of a symmetrical set of positive and negative values could involve considerable cost. In this situation a program is not manipulating representations details, but having to deal with the

integer types 303 sizes

consequences of a particular representation. Given the rarity of two of the possible integer representations being encountered, and overwhelming support for an asymmetric implementation of two's complement notation, the following is suggested:

- One's complement or sign and magnitude issues should only be dealt with when writing for such implementations.
- The asymmetric nature of existing two's complement implementations should be accepted and no attempts made to handle the *extra* negative value any differently than the other values.

---

615 In the case of sign and magnitude and one's complement, if this representation is a normal value it is called a *negative zero*. negative zero

Commentary

This defines the term *negative zero*. The term *zero* is commonly used to imply the zero value with the sign bit also set to zero. The term *positive zero* is sometimes used to help distinguish this case when discussing both representations of zero.

C90

The C90 Standard supported hosts that included a representation for negative zero, however, the term *negative zero* was not explicitly defined.

C++

The term *negative zero* does not appear in the C++ Standard.

Other Languages

This terminology is also used in Cobol, where negative zero is always supported irrespective of the underlying host representation of binary integers.

---

616 If the implementation supports negative zeros, they shall be generated only by: negative zero only generated by

Commentary

This is a requirement on the implementation. In the case of arithmetic operations, unless one of the operands has a value of negative zero, the operation does not deliver a negative zero result. The principle of *least surprise* (sometimes known as the *Principle of Least Astonishment*) could be said to be the driving principle here.

This requirement only applies to integer types and the list excludes all operators whose result is not derived by manipulating the operands (e.g., relational operators, comparison operators, or logical operators, which are all defined to return a value of 0 or 1). For these operators, an implementation cannot choose to use a negative zero value.

Negative zero can also be created as the result of a conversion operation. 686 floating-point converted to integer

C90

The properties of negative zeros were not explicitly discussed in the C90 Standard.

C++

This requirement was added in C99; negative zeros are not explicitly discussed in the C++ Standard.

Other Languages

Cobol also supports a negative zero under some conditions.

Example

```
1  _Bool is_negative_zero(int valu)
2  {
3  if (valu == 0)
4     if (value & ~0)
5        return 1;
6  return 0;
7  }
```

---

bitwise operators
negative zero

— the &, |, ^, ~, <<, and >> operators with arguments that produce such a value; 617

Commentary

The bitwise operators operate directly on the individual bits of the value representation of integer types. If negative zero is supported by an implementation, there is the possibility that any of them could generate the bit pattern representing this value (starting from a bit pattern that does not represent negative zero). The use of bitwise operators on values having a signed integer type and exhibit implementation-defined and undefined behaviors (Also the standard does not guarantee that a negative zero value can be stored into an object without a change to its representation.)

negative zero 615
bitwise operations 946
signed types
negative zero 620
storing

Other Languages

Cobol does not include these operators.

Common Implementations

The Unisys A Series[1390] uses signed magnitude representation. If the operands have unsigned types, the sign bit is not affected by these bitwise operators. If the operands have signed types, the sign bit does take part in bitwise operations.

Example

The following will produce a negative zero on an implementation that uses a sign and magnitude representation; for implementations that use other representations the behavior is undefined.

```
1  int unsigned_zero(void)
2  {
3  return (-1) & (-2);
4  }
```

— the +, -, *, /, and % operators where one argument is a negative zero and the result is zero; 618

Commentary

The + and - operators exist in unary and binary form. The phrase *where one argument* could be taken to imply two operands (i.e., only the binary operator are intended). In the case of unary minus this requirement means that -0 does not represent negative zero. In fact only negating a negative zero can return a result of negative zero.

negative zero 615

— compound assignment operators based on the above cases. 619

Commentary

This is specified to ensure that all the cases are covered. Compound assignment is defined in terms of its equivalent binary operator.

compound assignment 1312
semantics

Other Languages

Cobol does not contain compound assignment operators.

Common Implementations

Compound assignment is commonly implemented using the associated operator followed by an assignment. Although processors having instructions capable of operating directly on the contents of storage are no longer common (e.g., the Motorola 680x0 family[968]), implementations are not required to operate directly on the contents of storage for this permission to apply.

---

620 It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

negative zero storing

Commentary

It is unspecified because developers may not have any control over what happens and the combinations of circumstances may be too disparate for an implementation to document. For instance, in one case a value may be operated on in a register before being written to storage (so any conversion that may be carried out by the store will not have happened), while in another case the same value may be operated on directly in its storage location. It is possible that a store operation, treating the contents of a register as signed, may convert the value to another kind of zero.

C90

This unspecified behavior was not called out in the C90 Standard, which did not discuss negative integer zeros.

C++

Negative zero is not discussed in the C++ Standard.

Other Languages

Cobol specifies when a negative zero is generated. For the object types involved, there is never any question of a negative zero being converted to any other value when it is written to storage.

Common Implementations

The Unisys A Series[1390] uses a sign and magnitude representation and does not copy the sign bit when storing a value having an **`unsigned int`** type.

Coding Guidelines

The surprising effects of negative zeros is something that will need to be brought to developers' attention if a processor with this characteristic is ever encountered. However, implementations that support more than one integer representation of zero are very rare. For this reason no recommendations are made.

---

621 If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with arguments that would produce such a value is undefined.

Commentary

On processors that use a sign and magnitude representation the behavior of the code `(-1) & (-2)` will either be acceptable to an implementation or result in undefined behavior.

611 sign and magnitude
617 bitwise operators
negative zero

C90

This undefined behavior was not explicitly specified in the C90 Standard.

C++

This specification was added in C99 and is not explicitly specified in the C++ Standard.

---

622 The values of any padding bits are unspecified.[45)]

integer padding bit values

unsigned integer padding bit values 596

Commentary

This duplicates a sentence given earlier. The earlier sentence could be read to apply to unsigned integer types only. This sentence applies to all integer types.

---

object representation same padding signed/unsigned

A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. 623

Commentary

This is a requirement on the implementation. Two corresponding integer types must use the same number of bytes in their object representation. The value bits of a signed integer type are required to be a subset of the corresponding unsigned type and these value bits are at the same relative bit positions. If a signed integer object contains a positive value it can be copied, using `memcpy`, into an object having the corresponding unsigned integer type and the two value representations will compare equal. The signed integer type contains a sign bit, which may be a value bit in the corresponding unsigned integer type (in two's complement implementations), or it may not (in some sign and magnitude implementation— e.g., the Unisys A Series[1390]).

value bits 600 signed/unsigned positive 495 signed integer type subrange of equivalent unsigned type footnote 509 31 alignment 39

It is possible for the unsigned type to have more than one additional value bit. An earlier sentence deals with the value representation.

The interchangeability discussed in footnote 31 does not mention pointers to corresponding signed/unsigned integer types. The preceding requirement, along with the alignment requirements, is sufficient to deduce that pointers to corresponding signed/unsigned integer types can be used to access positive values held in the pointed-to objects.

C90

positive 495 signed integer type subrange of equivalent unsigned type

There was no such requirement on the object representation in C90, although this did contain the C99 requirement on the value representation.

C++

3.9.1p3 *. . . ; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.*

Example

This requirement guarantees that:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  signed short ss;
5  unsigned short us;
6
7  void f(void)
8  {
9  memcpy(&us, &ss, sizeof(ss));
10
11 if ((long)us == (long)ss)
12    printf("Value of ss is positive or (sizeof(short) == sizeof(long))\n");
13 }
```

is defined if the value of `ss` is positive.

For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type. 624

Commentary

This sentence was added by the response to DR #263.

DR #263

Problem

Consider the code:

```
int v [10];
memset (v, 0, sizeof v);
```

Most programmers would expect this code to set all the elements of `v` *to zero. However, the code is actually undefined: it is possible for* `int` *to have a representation in which all-bits-zero is a trap representation (for example, if there is an odd-parity bit in the value).*

C90

The C90 Standard did not specify this requirement.

---

625 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.

precision
integer type

Commentary

This defines the term *precision* (when applied to integer types). This term is also used in the definition of the I/O conversion specifiers. The term *precision* has a common usage meaning in the sense of the accuracy of a measurement. It is also used in the context of floating types. Values or objects having a floating type are commonly referred to as being single- or double-precision.

335 precision
floating-point

C90

The definition of this term is new in C99.

C++

The term *precision* was added in C99 and is only defined in C++ for floating-point types.

Other Languages

The term *precision* is used in several different languages to mean a variety of different things.

Coding Guidelines

The term *width* is often used by developers when discussing the number of bits used to represent integer types. Use of the term *precision*, in an integer type context is limited to only a handful of locations in the C Standard. There are no obvious advantages to training developers to use this term in the context of integer types.

626 width
integer type

---

626 The *width* of an integer type is the same but including any sign bit;

width
integer type

Commentary

This defines the term *width* (when applied to integer types), $width = precision + one\_if\_there\_is\_a\_sign\_bit$. If the width of type A is less than the width of type B, then type A is said to be *narrower* than type B.

narrower type
1306 narrower
example

C90

The definition of this term is new in C99.

C++

The term *width* was added in C99 and is not defined in the C++ Standard.

Other Languages

The term *width* is used in several different languages to mean a variety of different things.

Coding Guidelines

Many terms are commonly used by developers to describe the number of bits in an integer type, including the term *number of bits*. Because there is no commonly used term, guideline documents will need to fully define the terms used. The term *width* has the advantage of being defined by the C Standard and fitting with the usage of the term *narrow*.

---

thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision. 627

Commentary

rank 662 standard integer types

The standard does not require the precision of integer types having the same rank (i.e., a signed integer type and its equivalent unsigned form) to be the same. However, in practice they are the same in the majority of implementations.

---

footnote 45

45) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. 628

Commentary

footnote 606 44

This footnote duplicates footnote 44.

Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow. 629

Commentary

footnote 606 44

This footnote duplicates footnote 44.

All other combinations of padding bits are alternative object representations of the value specified by the value bits. 630

Commentary

footnote 606 44

This footnote duplicates footnote 44.

6.2.7 Compatible type and composite type

Commentary

This clause primarily deals with type compatibility across translation units. Few developers are aware of the concept of *composite type*. It is needed to handle the cases where there is more than one declaration of the same identifier in the same scope and name space.

Coding Guidelines

Translators are not required to perform type compatibility checks across translation units, and it is very rare to find one that does. Some static analysis tools perform various kinds of cross translation unit type checking.[681] Without automated tool support, these checks are unlikely to be carried out. The approach of these coding guideline is to recommend practices that remove the need to perform these checks, e.g., having a single, textually, point of declaration for identifiers.

identifier 422.1 declared in one file

---

compatible type if same type

Two types have *compatible type* if their types are the same. 631

Commentary

This defines the term *compatible type*. It is possible for two types to be compatible when their types are not the same type. When are two types the same? Other sentences in the standard specify when two types are the same.

qualified type 1494 to be compatible
pointer types 1562 to be compatible
array type 1585 to be compatible

C++

The C++ Standard does not define the term *compatible type*. It either uses the term *same type* or *different type*. The terms *layout-compatible* and *reference-compatible* are defined by the C++ Standard. The specification of layout-compatible structure (9.2p14) and layout compatible union (9.2p15) is based on the same set of rules as the C cross translation unit compatibility rules. The purpose of layout compatibility deals with linking objects written in other languages; C is explicitly called out as one such language.

Other Languages

All languages that have more than one type specify rules for how objects and values of different types may occur together as operands of operators. Languages in the Pascal family have a much stricter concept of type compatibility than C. It is possible for two types to be incompatible because their type names are different, even if their underlying representation is identical. Languages like Basic, Perl, and PL/1 take a much more laid-back approach, trying to provide conversions between almost any pair of types.

650 footnote 46

Coding Guidelines

Developers who have seen the advantages of a stricter definition of compatible type might be tempted to try to create guidelines that implement such a system; for instance, requiring name equivalence for types in C. However, an increase in strictness of the type system can only, practically, be enforced using automatic tool support. Developer training will also be required. Experience suggests that it takes months, if not a year or more, for some developers to be able to design and make use of good type categories. In:

type compatibility general guideline

```
1   typedef int frequency;
2   typedef int dpi;
3
4   extern frequency color_red;
5
6   dpi printer_resolution(char *printer_name)
7   {
8   /* ... */
9   return color_red;
10  }
```

the type of the expression returned by `printer_resolution` is compatible with the return type.

The same source rewritten in Ada would result in a diagnostic being issued; the type of the return expression is not compatible with the declared return type of the function. Yes, the developer-defined types, `frequency` and `dpi`, do have the same underlying basic type, but it makes no sense to return a frequency from this function. The expected return type is `dpi`.

It is believed that type checking of this kind, coupled with source code that makes extensive use of the appropriate type definitions, reduces the cost of software development (because in many cases unintended or ill-formed constructs are flagged, and can be corrected at translation time). However, there have been no studies investigating the cost effectiveness of strong typing.

---

632 Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.5 for declarators.[46)]

compatible type additional rules

Commentary

Another way of expressing the idea behind these rules is that compatible types always have the same representation and alignment requirements. The following are the five sets of type compatibility rules:

1. Same type.
2. Structure/union/enumerated types across translation units (the following C sentences).
3. Type specifier rules.
4. Type qualifier rules.

631 compatible type if
1378 type specifier syntax
1476 type qualifier syntax

declarator syntax 1547

5. Declarator rules.

compatible separate translation units

Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: 633

Commentary

These requirements apply if the structure or union type was declared via a typedef or through any other means. Because there can be more than one declaration of a type in the same translation unit, these requirements really apply to the composite type in each translation unit.

compos-ite type 642

In the following list of requirements, those that only apply to structures, unions, enumerations, or a combination thereof are explicitly called out as such. There is no requirement that prevents a structure type from being compatible with an appropriate union type; at the time of writing this issue is the subject of an outstanding DR (251). Two types are compatible if they obey both of the following requirements:

- *Tag compatibility.*
  1. If both types have tags, both shall be the same.
  2. If one, or neither, type has a tag, there is no requirement to be obeyed.
- *Member compatibility.* Here the requirement is that for every member in both types there is a corresponding member in the other type that has the following properties:
  1. The corresponding members have a compatible type.
  2. The corresponding members either have the same name or are unnamed.
  3. For structure types, the corresponding members are defined in the same order in their respective definitions.
  4. For structure and union types, the corresponding members shall either both be bit-fields having the same width, or neither shall be bit-fields.
  5. For enumerated types, the corresponding members (the enumeration constants) have the same value.

C90

Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types;

There were no requirements specified for tag names in C90. Since virtually no implementation performs this check, it is unlikely that any programs will fail to link when using a C99 implementation.

C++

The following paragraph applies when both translation units are written in C++.

3.2p5 *There can be more than one definition of a class type (clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (clause 14), non-static function template (14.5.5), static data member of a class template (14.5.1.3), member function template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.4) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements.*

There is a specific set of rules for dealing with the case of one or more translation units being written in another language and others being written in C++:

> *Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a* `linkage-specification`*:* 7.5p2
>
> . . .
>
> *[Note: . . . The semantics of a language linkage other than C++ or C are implementation-defined.]*

which appears to suggest that it is possible to make use of defined behavior when building a program image from components translated using both C++ and C translators.

The C++ Standard also requires that:

> *The* `class-key` *or* **`enum`** *keyword present in the* `elaborated-type-specifier` *shall agree in kind with the declaration to which the name in the* `elaborated-type-specifier` *refers.* 7.1.5.3p3

Other Languages

In many languages that support separate compilation, the implementation is required to store information on the types of identifiers defined in one unit that may be used in other units. This information is then read by a translator when already-translated units are referenced. Such a separate compilation model means that there is only ever one declaration of a type. (there is the configuration-management issue caused by changes to an externally visible type needing to be percolated through to any units that reference it through retranslation.)

Common Implementations

It is very rare for an implementation to perform cross translation unit checking of structure, union, or enumerated types. Most linkers (translation phase 8) simply match-up identifiers from different translation units, that have the same spelling. The extra checks performed by C++ implementations usually only apply to identifiers having function type (they are needed to support function overloading).

139 translation phase 8

Coding Guidelines

Having the same identifier declared in more than one source file opens the door to a modification of a declaration in one source file not being mirrored by equivalent changes in other source files. If there is only one source file containing the declaration, a developer header file, this problem disappears.

type definition only one
422.1 identifier declared in one file

Dev 422.1 The declaration of an incomplete structure or union type may occur in a header file, provided any subsequent completion of its type occurs in only one of the source files making up a complete program.

Example

file_1.c

```
1 union u { int m1 : 3; int : 8; int : 8; };
```

file_2.c

```
1 union u { int m1 : 3; int : 8;          }; /* Not compatible with declaration in file_1.c */
```

---

634 If one is declared with a tag, the other shall be declared with the same tag.

tag declared with same

Commentary

The same tag, as in the spelling of the tags, shall be identical (subject to translator limits on the length of internal identifiers). Nothing is said about members here because one or more of the types may be incomplete.

C90

This requirement is not specified in the C90 Standard.
Structures declared using different tags are now considered to be different types.

xfile.c

```
#include <stdio.h>

extern int WG14_N685(struct tag1 *, struct tag1 *);

struct tag1 {
            int m1,
                m2;
            } st1;

void f(void)
{
if (WG14_N685(&st1, &st1))
   {
   printf("optimized\n");
   }
else
   {
   printf("unoptimized\n");
   }
}
```

yfile.c

```
struct tag2 {
            int m1,
                m2;
            };
struct tag3 {
            int m1,
                m2;
            };

int WG14_N685(struct tag2 *pst1,
              struct tag3 *pst2)
{
pst1->m1 = 2;
pst2->m1 = 0;   /* alias? */

return pst1->m1;
}
```

An optimizing translator might produce **optimized** as the output of the program, while the same translator with optimization turned off might produce **unoptimized** as the output. This is because translation unit `y.c` defines `func` with two parameters each as pointers to different structures, and translation unit `x.c` calls `WG14_N685func` but passes the address of the same structure for each argument.

Other Languages

Very few other languages make use of the concept of tags.

Coding Guidelines

While the spelling of tag names might be thought not to alter the behavior of a program, the preceding example shows how a translator might rely on the information provided by tag names to make optimization decisions. While translators that take such information into account are rare, at the time of this writing, the commercial pressure to increase the quality of generated machine code means that such translators are likely

to become more common. The guideline recommendation dealing with a single point of declarations is applicable here.

422.1 identifier declared in one file

Another condition is where two unrelated structure or union types, in different translation units, use the same tag name. Identifiers in the tag name space have internal linkage and the usage may occur in a strictly conforming program. However, the issue is not language conformance but likelihood of developer confusion. The guideline recommendation dealing with reusing identifier names is applicable here.

792.3 identifier reusing names

Example

file_1.c

```
1  extern struct S {
2                  int m1;
3                  } x;
```

file_2.c

```
1  extern struct S {
2                  int m1;
3                  } x;
```

file_3.c

```
1  extern struct T {
2                  int m1;
3                  } x;
```

The declarations of x in `file_1.c` and `file_2.c` are compatible. The declaration in `file_3.c` is not compatible.

---

635 If both are complete types, then the following additional requirements apply:

Commentary

The only thing that can be said about the case where one type is complete and the other is incomplete is that their tag names match. There is no danger of mismatch of member names, types, or relative ordering with a structure if the type is incomplete. If both are complete, there is more information available to be compared.

Except in one case, a pointer to an incomplete structure type, a type used in the declaration of an object or function must be completed by the end of the translation unit that contains it.

---

636 there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types, and such that if one member of a corresponding pair is declared with a name, the other member is declared with the same name.

Commentary

For structure types, this requirement ensures that the same storage location is always interpreted in the same way when accessed through a specific member. It is also needed to ensure that the offsets of the storage allocated for successive members is identical for both types. For union types, this requirement ensures that the same named member is always interpreted in the same way when accessed.

What does *corresponding member* mean if there are no names to match against? The ordering rule is discussed elsewhere.

637 members corresponding

The standard does not give any explicit rule for breaking the recursion that can occur when checking member types for compatibility. For instance:

file_3.c

```
1  extern struct  S4 {
2                    struct S4 *mem;
3                    } glob;
```

file_4.c
```
1  extern struct  S4 {
2                     struct S4 *mem;
3                     } glob;
```

enumeration constant type 864

Enumeration constants always have an implied type of **int**.

The reply to DR #013 explains that the Committee feels the only reasonable solution is for the recursion to stop and the two types to be compatible.

C90

. . . if they have the same number of members, the same member names, and compatible member types;

The C90 Standard is lax in that it does not specify any correspondence for members defined in different structure types, their names and associated types.

C++

3.2p5 *— each definition of D shall consist of the same sequence of tokens; and*

— in each definition of D, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a ***const*** *object with internal or no linkage if the object has the same integral or enumeration type in all definitions of D, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of D; and*

The C Standard specifies an effect, compatible types. The C++ Standard specifies an algorithm, the same sequence of tokens (not preprocessing tokens), which has several effects. The following source files are strictly conforming C, but undefined behavior in C++.

file_1.c
```
1  extern struct {
2                short s_mem1;
3                } glob;
```

file_2.c
```
1  extern struct {
2                short int s_mem1;
3                } glob;
```

9.2p14 *Two POD-struct (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types (3.9).*

9.2p15 *Two POD-union (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types (3.9).*

Layout compatibility plays a role in interfacing C++ programs to other languages and involves types only. The names of members plays no part.

Other Languages

Those languages that have a more sophisticated model of separate compilation usually ensure that textually, the same sequence of tokens is seen by all separately translated source files.

Coding Guidelines

Using different names for corresponding members is unlikely to change the generated machine code. However, it could certainly be the cause of a great deal of developer confusion. The guideline recommendation specifying a single point of declaration is applicable here.

422.1 identifier declared in one file

Example

In the two files below the two declarations of x1 are compatible, but neither x2 or x3 are compatible across translation units.

file_1.c

```
1  extern struct S1 {
2                   int m1;
3                   } x1;
4  extern struct S2 {
5                   int m1;
6                   } x2;
7  extern struct S3 {
8                   int m1;
9                   } x3;
```

file_2.c

```
1  extern struct S1 {
2                   int m1;
3                   } x1;
4  extern struct S2 {
5                   long m1;
6                   } x2;
7  extern struct S3 {
8                   int m_1;
9                   } x3;
```

637 For two structures, corresponding members shall be declared in the same order.

members corresponding

Commentary

On first reading this sentence appears to be back to front; shouldn't the order decide what the corresponding members are? In fact, the wording is a constructive definition of the term *corresponding*. The standard defines a set of requirements that must be met, not an algorithm for what to do. It is the developer's responsibility to ensure that the requirements are met. In the case of unnamed bit-fields, this ordering requirement means there is only one way of creating a correspondence that meets them.

These requirements ensure that members of the same structure type have the same offsets within different definitions (in different translation units) of that structure type. There is no corresponding requirement on union types because all members start at the same address offset. The order in which union members are declared makes no difference to their member storage layout.

1207 pointer to union members compare equal
1354 storage layout

Enumeration constants may also be declared in different orders within their respective enumerated type definitions.

C90

. . . for two structures, the members shall be in the same order;

The C90 Standard is lax in that it does not specify how a correspondence is formed between members defined in different structure definitions. The following two source files could have been part of a strictly conforming program in C90. In C99 the behavior is undefined and, if the output depends on `glob`, the program will not be strictly conforming.

file_1.c

```
extern struct {
              short s_mem1;
              int i_mem2;
              } glob;
```

file_2.c

```
extern struct {
              int i_mem2;
              short s_mem1;
              } glob;
```

While the C90 Standard did not require an ordering of corresponding member names, developer expectations do. A diagnostic, issued by a C99 translator, for a declaration of the same object as a structure type with differing member orders, is likely to be welcomed by developers.

Coding Guidelines

identifier declared in one file 422.1

The guideline recommendation specifying a single point of declaration is applicable here. For each structure type, its member names are in their own unique name space. There is never an issue of other structure types, containing members using the same set of names, influencing other types.

Example

In the following example none of the members are not declared in the same order; however, it only matters in the case of structures.

file_1.c

```
extern struct S1 {
                  int m1;
                  char m2;
                  } x;
extern union U1 {
                 int m1;
                 char m2;
                 } y;

extern enum {E1, E2} z;
```

file_2.c

```
extern struct S2 {
                  char m2;
                  int m1;
                  } x;
extern union U1 {
                 char m2;
                 int m1;
                 } y;

extern enum {E2=1, E1=0} z;
```

638 For two structures or unions, corresponding bit-fields shall have the same widths.

Commentary

The common initial sequence requirement ensures that the decision on where to allocate bit-fields within a storage unit is deterministic. If corresponding bit-fields have the same width they will be placed in the same location, relative to other members, in both structure types. In the case of union types a bit-field can occur in any position within the lowest address storage unit. Requiring that the members have the same width ensures that implementations assign them the same bit offsets within the storage unit.

1038 common initial sequence

Common Implementations

Implementations assign locations within storage units to bit-fields based on their width (and type if non-**int** bit-fields are supported as an extension). These factors are dealt with by these type compatibility requirements.

1395 bit-field shall have type

Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here.

422.1 identifier declared in one file

639 For two enumerations, corresponding members shall have the same values.

Commentary

The only requirement is on the value, not on how that value was specified in the source. For instance, it could have been obtained through a direct assignment of any number of constant expressions (all returning the same value), or implicitly calculated to be one greater than the previous enumeration constant value.

C90

> *. . . for two enumerations, the members shall have the same values.*

The C90 Standard is lax in not explicitly specifying that the members with the same names have the same values.

C++

> *— each definition of D shall consist of the same sequence of tokens; and* 3.2p5

The C++ requirement is stricter than C. In the following two translation units, the object `e_glob` are not considered compatible in C++:

file_1.c
```
1 extern enum {A = 1, B = 2} e_glob;
```

file_2.c
```
1 extern enum {B= 2, A = 1} e_glob;
```

Common Implementations

The debugging information written out to the object file in some implementations uses an implied enumeration constant ordering (i.e., that given in the definition). This can cause some inconsistencies in the display of enumeration constants, but debuggers are outside the scope of the standard.

Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here.

422.1 identifier declared in one file

Example

In the following two files members having the same name appear in the same order, but their values are different.

file_1.c
```
extern enum {E1, E2} z;
```

file_2.c
```
extern enum {E1 = 99, E2} z;
```

---

same object have compatible types
same function have compatible types

All declarations that refer to the same object or function shall have compatible type; 640

Commentary

This *shall* is not in a constraint, making it undefined behavior if different declarations of the same object are not compatible types. There can be more than one declaration referring to the same object or function if

- they occur in different translation units, in which case they will have external linkage;
- they occur in the same translation unit with internal linkage, or external linkage.

no linkage 424
identifier declaration is unique

It is not possible to have multiple declarations of identifiers that have no linkage. The case:

file_1.c
```
extern int i;
```

file_2.c
```
extern enum {e1, e2} i;
```

(assuming that this enumerated type is compatible with the type **int**) is something of an oddity.

C++

3.2p5 — *each definition of D shall consist of the same sequence of tokens; and*

— *in each definition of D, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a* ***const*** *object with internal or no linkage if the object has the same integral or enumeration type in all definitions of D, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of D; and*

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4).*

The C++ Standard is much stricter in requiring that the types be identical. The **int**/**enum** example given above would not be considered compatible in C++. If translated and linked with each other the following source files are strictly conforming C, but undefined behavior in C++.

file_1.c
```
extern short es;
```

file_2.c
```
extern short int es = 2;
```

Other Languages

Languages that have a type system usually require that declarations of the same object or function, in separately translated source files, have compatible (however that term is defined) types.

Common Implementations

It is very rare for an implementation to do cross translation unit type compatibility checking of object and function declarations. C++ implementations use some form of name mangling to resolve overloaded functions (based on the parameter types). Linkers supporting C++ usually contain some cross translation unit checks on function types. If C source code is translated by a C++ translator running in a C compatibility mode, many of these link-time checks often continue to be made.

Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here. Within existing code, old-style function declarations often still appear in headers. Such usage does provide a single declaration. However, these declarations do not provide all of the type information that it is possible to made available. For economic reasons developers may choose to leave these existing declarations unchanged.

422.1 identifier declared in one file

Example

Most implementations will translate and create a program image of the following two files without issuing any diagnostic:

file_1.c

```
extern int f;

int main(void)
{
f++;
return 0;
}
```

file_2.c

```
extern int f(void)
{
return 0;
}
```

641 otherwise, the behavior is undefined.

Commentary

The most common behavior is to create a program image without issuing a diagnostic. The execution-time behavior of such a program image is very unpredictable.

C++

A violation of this rule on type identity does not require a diagnostic. 3.5p10

The C++ Standard bows to the practical difficulties associated with requiring implementations to issue a diagnostic for this violation.

Other Languages

Some languages require that a diagnostic be issued for this situation. Others say nothing about how their translators should behave in this situation.

Common Implementations

The linkers in most implementations simply resolve names without getting involved in what those names refer to. In many implementations storage for all static duration objects is aligned on a common boundary (e.g., a two- or eight-byte boundary). This can mean, for scalar types, that there is sufficient storage available, whatever different types the same object is declared with. (Linkers usually obtain information on the number of bytes in an object from the object-file containing its definition, ignoring any such information that might be provided in the object-files containing declarations of it.) The effect of this link-time storage organization strategy is that having the same object declared with different scalar types, in different translation units, may not result in any unexpected behaviors (other objects being modified as a side-effect of assigning to the object) occurring.

---

composite type

A *composite type* can be constructed from two types that are compatible; 642

Commentary

This C paragraph defines the term *composite type*; which are rarely talked about outside of C Standards-related discussions. Composite types arise because C allows multiple declarations, in some circumstances, of the same object or function. Two declarations of the same identifier only need be compatible; neither are they required to be token-for-token identical. In practice composite types are applied to types occurring in the same translation unit. There is no construction of composite types across translation units.

C++

One of the two types involved in creating composite types in C is not supported in C++ (function types that don't include prototypes) and the C++ specification for the other type (arrays) is completely different from C.

array 644 composite type
conditional 1282 operator pointer to qualified types

Because C++ supports operator overloading type qualification of pointed-to types is a more pervasive issue than in C (where it only has to be handled for the conditional operator). The C++ Standard defines the concept of a *composite pointer type* (5.9p2). This specifies how a result type is constructed from pointers to qualified types, and the null pointer constant and other pointer types.

Other Languages

In most languages two types can only be compatible if they are token-for-token identical, so the need for a composite type does not usually occur.

Coding Guidelines

identifier 422.1 declared in one file

If the guideline recommendation specifying a single point of declarations is followed, a composite type will only need to be created in one case— in the translation unit that defines an object, or function, having external linkage.

Example

Assume the type **int** is the type that all enumerated types are compatible with:

```
1  enum ET {r, w, b};
2
3  extern enum ET obj;
4  extern int obj;
```

Possible composite type are the enumeration, ET, or the basic type **int**.

---

it is a type that is compatible with both of the two types and satisfies the following conditions: 643

Commentary

A composite type is formed by taking all the available information from both type declarations to create a type that maximizes what is known. The specification in the standard lists the properties of the composite type in relation to the two types from which it is created. It is not a set of rules to follow in constructing such a type. The possible composite types meeting the specification is not always unique.

644 — If one type is an array of known constant size, the composite type is an array of that size;

array composite type

Commentary

For them to be compatible, the other array would have to be an incomplete array type. In this case encountering a declaration of known constant size also completes the type.

548 known constant size

C90

If one type is an array of known size, the composite type is an array of that size;

Support for arrays declared using a nonconstant size is new in C99.

C++

An incomplete array type can be completed. But the completed type is not called the composite type, and is regarded as a different type:

. . . ; the array types at those two points ("array of unknown bound of T" and "array of N T") are different types. 3.9p7

The C++ Standard recognizes the practice of an object being declared with both complete and incomplete array types with the following exception:

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). 3.5p10

645 otherwise, if one type is a variable length array, the composite type is that type.

Commentary

This situation can only occur if the array type appears in function prototype scope. In this scope a VLA type is treated as if the size was replaced by *. The case of one of the array types having a constant size is covered by the previous rule; the remaining possibilities are another VLA type or an incomplete array type. In both cases a VLA type will be compatible with them, providing their element types are compatible.

1581 VLA size treated as *
1585 array type to be compatible

C90

Support for VLA types is new in C99.

C++

Variable length array types are new in C99. The C++ library defined container classes (23), but this is a very different implementation concept.

Example

The composite type of:

```
1  extern int i(int m, int p4[m]);
2  extern int i(int n, int p4[n]);
```

is:

```
1  extern int i(int m, int p4[*]); /* Parameter names are irrelevant. */
```

function composite type

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list. 646

Commentary

The other type could be an old-style function declaration. If the other type is also a function prototype, the compatibility requirements mean that additional information can only be provided if the parameters have a pointer-to function type (which is an old-style function declaration). This issue is discussed in more detail elsewhere.

parameter 1616 qualifier in composite type

C++

All C++ functions must be declared using prototypes. A program that contains a function declaration that does not include parameter information is assumed to take no parameters.

```
1  extern void f();
2
3  void g(void)
4  {
5  f(); // Refers to a function returning int and having no parameters
6       /* Non-prototype function referenced */
7  }
8
9  void f(int p) /* Composite type formed, call in g linked to here */
10                // A different function from int f()
11                // Call in g does not refer to this function
12 { /* ... */ }
```

Other Languages

Most languages either require that the types of parameters be given or that the types not be given. C is unusual in allowing both kinds of function declarations. Both C++ and Java require that the parameter types always be given.

— If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters. 647

Commentary

This is a recursive definition. If a parameter has pointer-to function type, then a composite type will be constructed for its parameters, and so on. Wording elsewhere in the standard specifies that the composite type of parameters is the unqualified version of their types.

parameter 1616 qualifier in composite type

C++

C++ allows functions to be overloaded based on their parameter types. An implementation must not form a composite type, even when the types might be viewed by a C programmer as having the same effect:

```
1  /*
2   * A common, sloppy, coding practice.  Don't declare
3   * the prototype to take enums, just use int.
4   */
5  extern void f(int);
6
7  enum ET {E1, E2, E3};
8
9  void f(enum ET p) /* composite type formed, call in g linked to here */
10                   // A different function from void f(int)
11                   // Call in g does not refer here
12 { /* ... */ }
13
14 void g(void)
```

```
{
f(E1); // Refers to a function void (int)
       /* Refers to definition of f above */
}
```

Example

It is thus possible to slowly build up a picture of what the final function prototype is:

```
void f();
void f(int p1[], const int p2, float *          p3);
void f(int p1[2],      int p2, float *          p3);
/*
 * Composite type is: void f(int p1[2], int p2, float *p3);
 */
void f(int p1[], const int p2, float * volatile p3);
/*
 * Composite type is unchanged.
 */
```

---

648 These rules apply recursively to the types from which the two types are derived.

Commentary

The possible types, not already covered before, to which these rules can be applied are array, structure, union, and pointer. Although it is not possible to declare an array of incomplete type, an array of pointer-to functions would create a recursive declaration that would need to be processed by these rules.

C++

The C++ Standard has no such rules to apply recursively.

Other Languages

Languages that support nesting of derived types usually apply any applicable rules recursively.

---

649 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[47)] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

prior declaration visible

Commentary

Any change to an object's type, to a composite type, will not affect the way that accesses to it are handled. Any operation that accesses an object after its first declaration, but before a second declaration, will behave the same way after any subsequent declaration is encountered.

It is possible for references to an identifier, having a function type, to be different after a subsequent declaration. This occurs if the type of a parameter is not compatible with its promoted type (the issue of the composite type of function types is also discussed elsewhere). For instance:

675 integer promotions
1616 parameter qualifier in composite type

```
extern void f();

void g_1(void)
{
f('w');
}

extern void f(char);

void g_2(void)
{
f('x');
}
```

function definition ends with ellipsis 1011

the first call to `f` uses the information provided by the first, old-style, declaration of `f`. At the point of this first call, the behavior is undefined (assuming the function definition is defined using a prototype). At the point of the second call to `f`, in `g_2`, the composite type matches the definition and the behavior is defined.

C90

The wording in the C90 Standard:

> *For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.*

was changed to its current form by the response to DR #011, question 1.

C++

3.5p9 *Two names that are the same (clause 3) and that are declared in different scopes shall denote the same object, reference, function, type, enumerator, template or namespace if*

— both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and

— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

— when both names denote functions, the function types are identical for purposes of overloading; and

This paragraph applies to names declared in different scopes; for instance, file scope and block scope externals.

13p1 *When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only function declarations can be overloaded; object and type declarations cannot be overloaded.*

The following C++ requirement is much stricter than C. The types must be the same, which removes the need to create a composite type.

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.*

The only composite type in C++ are composite pointer types (5.9p2). These are only used in relational operators (5.9p2), equality operators (5.10p2, where the term common type is used), and the conditional operator (5.16p6). C++ composite pointer types apply to the null pointer and possibly qualified pointers to **`void`**.

If declarations of the same function do not have the same type, the C++ link-time behavior will be undefined. Each function declaration involving different adjusted types will be regarded as referring to a different function.

```
1  extern void f(const int);
2  extern void f(int);        /* Conforming C, composite type formed */
3                             // A second (and different) overloaded declaration
```

Example

The following illustrates various possibilities involving the creation of composite types (the behavior is undefined because the types are not compatible with each other).

```
1   extern int a[];
2
3   void f(void)
4   {
5   extern int a[3]; /* Composite type applies inside this block only. */
6   }
7
8   void g(void)
9   {
10  extern int a[4]; /*
11                    * Compatible with file scope a[],
12                    * composite type applies to this block.
13                    */
14  }
15
16  void h(void)
17  {
18  float a;
19     {
20     extern int a[5]; /* No prior declaration visible. */
21     }
22  }
23
24  extern int a[5]; /* Composite type is int a[5]. */
```

650 46) Two types need not be identical to be compatible.

footnote 46

Commentary

Neither do two types have to consist of the same sequence of tokens to be the same types. It is possible for different sequences of tokens to represent the same type. The standard does not specify what it means for two types to be identical.

1380 type specifiers sets of

C++

The term *compatible* is used in the C++ Standard for *layout-compatible* and *reference-compatible*. Layout compatibility is aimed at cross-language sharing of data structures and involves types only. The names of structure and union members, or tags, need not be identical. C++ reference types are not available in C.

Other Languages

Languages take different approaches to the concept of compatibility. Pascal and many languages influenced by it use name equivalence (every type is unique and denoted by its name). Other languages, for instance CHILL, support *structural equivalence* (two types are the same if the underlying representation is the same, no matter how the declarations appear in the text). In fact CHILL specifies a number of different kinds of equivalence, which are used in different contexts.

structural equivalence

```
1   /*
2    * Some languages specify that the first two definitions of X are not
3    * compatible, because the structure of their declarations is different.
4    * Languages that do not expose the internal representation of a type
5    * could regard the last definition of X as being compatible with the
6    * previous two.  There are matching members and their types are the same.
7    */
8   struct X {
9             int a,
```

```
                b;
          };
struct X {
           int a;
           int b;
          };
struct X {
           int b;
           int a;
          };

/*
 * From the structural point of view the following
 * declarations are all different.
 */
extern unsigned int ui_1;
extern unsigned ui_2;
extern int unsigned ui_3;
```

Some languages (e.g., Ada, Java, and Pascal) base type equivalence on names, hence this form is called *name equivalence*. The extent to which name equivalence is enforced for arithmetic types varies between languages. Some languages are strict, while others have a more relaxed approach (Ada offers the ability to specify which approach to use in the type declaration).

```
typedef unsigned int UINT_1;
typedef unsigned int UINT_2;

/*
 * Some languages would specify that the following objects were not
 * compatible because they were declared with types that had different
 * names (an anonymous one in the last case).  Others would be more
 * relaxed, because the underlying types were the same.
 */
extern UINT_1 ui_1;
extern UINT_2 ui_2;
extern unsigned int ui_3;

/*
 * All languages based on name equivalence would treat the following
 * as different types that were not compatible with each other.
 */
typedef struct {
                int x, y;
               } coordinate;
typedef struct {
                int x, y;
               } position;
typedef int APPLES;
typedef int ORANGES;
```

Coding Guidelines

type compatibility 631
general guideline

Most coding guideline documents recommend a stricter interpretation of *identical* types than that made by the C Standard.

---

footnote 47

47) As specified in 6.2.1, the later declaration might hide the prior declaration. 651

Commentary

outer scope 413
identifier hidden

To be exact a different declaration, in a scope between that of the file scope declaration and the current declaration, might hide the later declaration.

652 EXAMPLE Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

C++

The C++ language supports the overloading of functions. They are overloaded by having more than one declaration of a function with the same name and return type, but different parameter types. In C++ the two declarations in the example refer to different versions of the function `f`.

6.3 Conversions

653 Several operators convert operand values from one type to another automatically.

operand convert automatically

Commentary

The purpose of these implicit conversions is to reduce the number of type permutations that C operators have to deal with. Forcing developers to use explicit casts was considered unacceptable. There is also the practical issue that most processors only contain instructions that can operate on a small number of scalar types.

Other Languages

There are two main lines of thinking about binary operators whose operands have different types. One is to severely restrict the possible combination of types that are permitted, requiring the developer to explicitly insert casts to cause the types to match. The other (more developer-friendly, or the irresponsible approach, depending on your point of view) is to have the language define implicit conversions, allowing a wide range of differing types to be operated on together.

Common Implementations

Conversion of operand values on some processors can be a time-consuming operation. This is one area where the as-if rule can be applied to make savings— for instance, if the processor has instructions that support operations on mixed types.

Coding Guidelines

Experience suggests that implicit conversions are a common cause of developer miscomprehension and faults in code. Some guideline documents only consider the problems caused by these implicit conversions and simply recommend that all conversions be explicit. The extent to which these explicit conversions increase the cost of program comprehension and are themselves a source of faults is not considered.

The first issue that needs to be addressed is why the operands have different types in the first place. Why weren't they declared with the same type? C developers are often overly concerned with using what they consider to be the *right* integer type. This usually involves attempting to use the type with the smallest representable range needed for the job. Using a single integer type would remove many implicit conversions, and is the recommended practice.

480.1 object int type only

How should those cases where the operands do have different types be handled? Unfortunately, there is no published research on the effects, on developer source code comprehension performance, of having all conversions either implicit or explicit in the source. Without such experimental data, it is not possible to make an effective case for any guideline recommendation.

The potential costs of using explicit conversions include:

- Ensuring that the types used in conversions continue to be the correct ones when code is modified. For instance, the expression `(unsigned int)x+y` may have been correct when the source was originally written, but changes to the declaration of `y` may need to be reflected in changes to the type used to

typedef name syntax 1629

cast x. Typedef names offer a partial solution to the dependence between the declaration of an object and its uses in that they provide a mechanism for dealing with changes in x's type. However, handling changes to both y's and y's types is more complex.

- An explicit conversion has to be explicitly read and comprehended, potentially increasing a reader's cognitive load for that expression. Depending on the reason for reading the expression, information on conversions taking place may represent a level of detail below what is required (readers are then paying a cost for no benefit). For instance, in the expression `(unsigned int)x+y` knowing that `x` and `y` are referenced together may be the only information of interest to a reader. The fact that they are added together may be the next level of detail of interest. The exact sequence of operations carried out in the evaluation of the expression being the final level of detail of interest.

The potential costs of relying on implicit conversions include:

- Modifications to the source may change the implicit conversions performed, resulting in a change of program behavior. While this situation is familiar to developers, it can also occur when explicit conversions are used. The implicit conversion case is more well known because most source relies on them rather than explicit conversions. It is not known whether the use of explicit conversions would result in a decrease or increase in faults whose root cause was conversion-related.
- In those cases where readers need to comprehend the exact behavior of an expression's evaluation, effort will need to be invested in deducing what conversions are performed. In the case of explicit conversions, it is generally assumed that the conversions specified are correct and it is often also assumed that they are the only ones that occur. No effort has to be expended in recalling operand types from (human) memory, or searching the source to obtain this information.

There is no experimental evidence showing that use of explicit conversions leads to fewer faults. Those cases where a potential benefit from using an explicit conversion might be found, include:

- Where it is necessary to comprehend the details of the evaluation of an expression, the cognitive effort needed to deduce the conversions performed will be reduced. It is generally assumed that the result of explicit conversions will not themselves be the subject of any implicitly conversions, and that the result type will be that of the explicit conversion.
- Experience has shown that developer beliefs about what implicit conversions will occur are not always the same as what the standard specifies. For instance, a common misconception is that no implicit conversion takes places if the operands have exactly the same type. Your author has heard several explanations for why a conversion is unnecessary, but has never recorded these (so no analysis is possible). A good example is the following:

```
1  unsigned char c1, c2, c3;
2
3  if (c1 + c2 > c3)
```

integer promotions 675

A developer who believes that no conversions take place will be under the impression that `c1+c2` will always return a result that is within the range supported by the type **`unsigned char`**. In fact, the integer promotions are applied to all the operands and the result of `c1+c2` can exceed the range supported by the type **`unsigned char`** (assuming the type **`int`** has a greater precision than the type **`unsigned char`**, which is by far the most common situation).

The following is one potential benefit of using implicit conversions:

- In those cases where exact details on the evaluation of an expression is not required, a reader of the source will not have to invest cognitive effort in reading and comprehending any explicit conversion operations.

The faults associated with mixing signed and unsigned operands are often perceived as being the most common conversion faults. Whether this perception is caused by the often-dramatic nature of these faults, or reflects actual occurrences, is not known.

Usage

Usage information on the cast operator is given elsewhere (see Table 1134.1).

Table 653.1: Occurrence of implicit conversions (as a percentage of all implicit conversions; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Converted to	Converted from	%	Converted to	Converted from	%
`( unsigned char )`	`_int`	33.0	`( int )`	`unsigned short`	1.9
`( unsigned short )`	`_int`	17.7	`( unsigned long )`	`_int`	1.8
(other-types)	other-types	11.3	`( unsigned int )`	`int`	1.7
`( short )`	`_int`	7.6	`( short )`	`int`	1.7
`( unsigned int )`	`_int`	5.1	`( enum )`	`_int`	1.3
(ptr-to)	ptr-to	4.7	`( unsigned long )`	`int`	1.2
`( char )`	`_int`	3.6	`( int )`	`char`	1.2
(ptr-to)	_ptr-to	2.9	`( int )`	`enum`	1.0
`( int )`	`unsigned char`	2.3			

654 This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*).

implicit conversion
explicit conversion

Commentary

This defines the terms *implicit conversion* and *explicit conversion*. The commonly used developer term for *implicit conversion* is *implicit cast* (a term that is not defined by the standard).

C++

The C++ Standard defines a set of implicit and explicit conversions. Declarations contained in library headers also contain constructs that can cause implicit conversions (through the declaration of constructors) and support additional explicit conversions— for instance, the complex class.

The C++ language differs from C in that the set of implicit conversions is not fixed. It is also possible for user-defined declarations to create additional implicit and explicit conversions.

Other Languages

Most languages have some form of implicit conversions. Strongly typed languages tend to minimize the number of implicit conversions. Other languages (e.g., APL, Basic, Perl, and PL/1) go out of their way to provide every possible form of implicit conversion. PL/1 became famous for the extent to which it would go to convert operands that had mismatched types. Some languages use the term *coercion*, not conversion. A value is said to be coerced from one type to another.

Coding Guidelines

Although translators do not treat implicit conversions any different from explicit conversions, developers and static analysis tools often treat them differently. The appearance of an explicit construct is given greater weight than its nonappearance (although an operation might still be performed). It is often assumed that an explicit conversion indicates that its consequences are fully known to the person who wrote it, and that subsequent readers will also comprehend its consequences (an implicit conversion is not usually afforded such a status). The possibility that subsequent modifications to the source may have changed the intended behavior of the explicit conversion, or that the developer may not have only had a limited grasp of the effects of the conversion (but just happened to work for an organization that enforces a *no implicit conversion* guideline), is not usually considered.

The issues involved in conversions between integer types is discussed elsewhere. The following example highlights some of the issues:

486 signed integer
corresponding unsigned integer

```
#if MACHINE_Z
typedef unsigned int X;
typedef signed int Y;
#else
typedef signed int X;
typedef unsigned int Y;
#endif

unsigned int ui;
signed int si;
X xi;
Y yi;

void f(void)
{
/*
 * It is not apparent from the source that the type of any object might change.
 */
unsigned int uloc;
signed int sloc;

uloc = ui + si;    uloc = ui + (unsigned int)si;
sloc = ui + si;    sloc = (signed int)(ui + si);
}

void g(void)
{
/*
 * The visible source shows that it is possible for the type of an object to change.
 */
X xloc;
Y yloc;

/*
 * The following sequence of casts might not be equivalent to those used in f.
 */
xloc = xi + yi;    xloc = xi + (X)yi;
yloc = xi + yi;    yloc = (Y)(xi + yi);
}
```

In the function `f`, the two assignments to `uloc` give different impressions. The one without an explicit cast of `si` raises the issue of its value always being positive. The presence of an explicit cast in the second cast gives the impression that the author of the code intended the conversion to take place and that there is no need to make further checks. In the second pair of assignments the result is being converted back to a signed quantity. The explicit cast in the second assignment gives the impression that the original author of the code intended the conversion to take place and that there is no need to make further checks.

In the function `g`, the issues are less clear-cut. The underlying types of the operands are not immediately obvious. The fact that typedef names have been used suggests some intent to hide implementation details. Is the replacement of a subset of the implicit conversions by explicit casts applicable in this situation? Having the definition of the typedef names conditional on the setting of a macro name only serves to reduce the possibility of being able to claim any kind of developer intent. Having a typedef name whose definition is conditional also creates problems for tool users. Such tools tend to work by tracing a single path of translation through the source code. In the function `g`, they are likely to give diagnostics applicable to one set of the `X` and `Y` definitions. Such diagnostics are likely to be different if the conditional inclusion results in a different pair of definitions for these typedef names.

Conversions between integer and floating types are special for several reasons:

- Although both represent numerical quantities, in the floating-point case literals visible in the source code need not be exact representations of the value used during program execution.

- The internal representations are usually significantly different.
- One of the representations is capable of representing very large and very small quantities (it has an exponent and fractional part),

These differences, and the resulting behavior, are sufficient to want to draw attention to the fact that a conversion is taking place. The issues involved are discussed in more detail elsewhere.

686 floating-point converted to integer
688 integer conversion to floating

Example

```
1  extern short es_1, es_2, es_3;
2
3  void f(void)
4  {
5  es_1 = es_2 + es_3;
6  es_1 = (short)((int)es_2 + (int)es_3);
7  es_1 = (short)(es_2 + es_3);
8  }
```

---

655 The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators;

Commentary

All operators might be said to be *ordinary* and these conversions apply to most of them (e.g., the shift operators are one example of where they are not applied).

702 operators cause conversions
1183 shift operator integer promotions

C++

Clause 4 ‘Standard conversions’ and 5p9 define the conversions in the C++ Standard.

---

656 it is supplemented as required by the discussion of each operator in 6.5.

Commentary

Clause 6.5 deals with expressions and describes each operator in detail.

940 expressions

C++

There are fewer such supplements in the C++ Standard, partly due to the fact that C++ requires types to be the same and does not use the concept of compatible type.

C++ supports user-defined overloading of operators. Such overloading could change the behavior defined in the C++ Standard, however these definitions cannot appear in purely C source code.

---

657 Conversion of an operand value to a compatible type causes no change to the value or the representation.

compatible type conversion

Commentary

This might almost be viewed as a definition of compatible type. However, there are some conversions that cause no change to the value, or the representation, and yet are not compatible types; for instance, converting a value of type **int** to type **long** when both types have the same value representation.

C++

No such wording applied to the same types appears in the C++ Standard. Neither of the two uses of the C++ term *compatible* (layout-compatible, reference-compatible) discuss conversions.

Other Languages

More strongly typed languages, such as Pascal and Ada, allow types to be created that have the same representation as a particular integer type, but are not compatible with it. In such languages, conversions are about changes of type rather than changes of representation (which is usually only defined for a few special cases, such as between integer and floating-point).

Common Implementations

While an obvious optimization for a conversion of an operand value to a compatible type is to not generate any machine code, some translators have been known to generate code to perform this conversion. The generation of this conversion code is often a characteristic of translators implemented on a small budget.

Coding Guidelines

Conversion of an operand value to a compatible type can occur for a variety of reasons:

- The type of the operand has changed since the code was originally written— the conversion has become redundant.
- The type specified in the explicit conversion is a typedef name. The definition of this typedef name might vary depending on the host being targeted. For some hosts it happens to be compatible with the type of the operand value; in other cases it is not.
- The cast operation, or its operand is a parameter in a function-like macro definition. In some cases the arguments specified may result in the expanded body containing a conversion of a value having a compatible type.

macro function-like 1933

In only the first of these cases might there be a benefit in removing the explicit conversion. The issue of redundant code is discussed elsewhere.

redundant code 190

Forward references: cast operators (6.5.4). 658

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

---

conversion rank

Every integer type has an *integer conversion rank* defined as follows: 659

Commentary

This defines the term *integer conversion rank*. Although this is a new term, there already appears to be a common usage of the shorter term *rank*. C90 contained a small number of integer types. Their relative properties were enumerated by naming each type with its associated properties. With the introduction of two new integer types and the possibility of implementation-defined integer types, documenting the specification in this manner is no longer practical.

The importance of the rank of a type is its value relative to the rank of other types. The standard places no requirements on the absolute numerical value of any rank. The standard does not define any mechanism for a developer to control the rank assigned to any extended types. That is not to say that implementations cannot support such functionality, provided the requirements of this clause are met.

There is no rank defined for bit-fields.

C90

The concept of integer conversion rank is new in C99.

C++

The C++ Standard follows the style of documenting the requirements used in the C90 Standard. The conversions are called out explicitly rather than by rank (which was introduced in C99). C++ supports operator overloading, where the conversion rules are those of a function call. However, this functionality is not available in C.

Other Languages

Most languages have a single integer type, which means there are no other integer types to define a relationship against. Languages that support an unsigned type describe the specific relationship between the signed and unsigned integer type.

Common Implementations

Whether implementations use the new concept of rank or simply extend the existing way things are done internally is an engineering decision, which is invisible to the user of a translator product — the developer.

Coding Guidelines

Although integer conversion rank is a new term it has the capacity to greatly simplify discussions about the integer types. Previously, general discussions on integer types either needed to differentiate them based on some representation characteristic, such as their size or width, or by enumerating the types and their associated attributes one by one.

---

660 — No two signed integer types shall have the same rank, even if they have the same representation.

Commentary

This mirrors the concept that two types are different, even if they have the same representation.

508 types different even if same representation

---

661 — The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

rank signed integer vs less precision

Commentary

The standard could equally well have defined things the other way around, with the rank being less than. On the basis that the precision of integral types continues to grow, over time, it seems more intuitive to also have the ranks grow. So the largest rank is as open-ended as the largest number of bits in an integral type. Given the requirement that follows this one, for the time being, this requirement, only really applies to extended integer types.

C++

The relative, promotion, ordering of signed integer types defined by the language is called out explicitly in clause 5p9.

---

662 — The rank of **`long long int`** shall be greater than the rank of **`long int`**, which shall be greater than the rank of **`int`**, which shall be greater than the rank of **`short int`**, which shall be greater than the rank of **`signed char`**.

rank standard integer types

Commentary

This establishes a pecking order among the signed integer types defined by the standard.

C++

Clause 5p9 lists the pattern of the usual arithmetic conversions. This follows the relative orderings of rank given here (except that the types **`short int`** and **`signed char`** are not mentioned; nor would they be since the integral promotions would already have been applied to operands having these types).

---

663 — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

rank corresponding signed/unsigned

Commentary

The only unsigned integer type that does not have a corresponding signed integer type is **`_Bool`**.

487 standard unsigned integer

Rank is not sufficient to handle all conversion cases. Information on the sign of the integer type is also needed.

488 extended unsigned integer

---

664 — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

rank standard integer relative to extended

Commentary

This is a requirement on the implementation. It gives preference to converting to the standard defined type rather than any extended integer type that shares the same representation.

Why would a vendor provide an extended type that is the same width as one of the standard integer types? The translator vendor may support a variety of different platforms and want to offer a common set of typedefs, across all supported platforms, in the **<stdint.h>** header. This could have the effect, on some platforms, of an extended integer type having the same width as one of the standard integer types. A vendor may also provide more than one representation of integer types. For instance, by providing support for extended integer types whose bytes have the opposite endianness to that of the standard integer types.

endian 570

C++

The C++ Standard specifies no requirements on how an implementation might extend the available integer types.

---

char rank

— The rank of **char** shall equal the rank of **signed char** and **unsigned char**. 665

Commentary

This statement is needed because the type **char** is distinct from that of the types **signed char** and **unsigned char**.

char 537 separate type

_Bool rank

— The rank of **_Bool** shall be less than the rank of all other standard integer types. 666

Commentary

This does not imply that the object representation of the type **_Bool** contains a smaller number of bits than any other integer type (although its value representation must).

unsigned 593 integer types object representation

C++

3.9.1p6 *As described below, **bool** values behave as integral types.*

4.5p4 *An rvalue of type **bool** can be converted to an rvalue of type **int**, with **false** becoming zero and **true** becoming one.*

The C++ Standard places no requirement on the relative size of the type **bool** with respect to the other integer types. An implementation may choose to hold the two possible values in a single byte, or it may hold those values in an object that has the same width as type **long**.

Other Languages

Boolean types, if supported, are usually viewed as the smallest type, irrespective of the amount of storage used to represent them.

---

rank enumerated type

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2). 667

Commentary

The compatible integer type can vary between different enumerated types. An enumeration constant has type **int**. There is no requirement preventing the rank of an enumerated type from being less than, or greater than, the rank of **int**.

enumeration 1447 type compatible with enumerators 1441 type int

Other Languages

Most languages that contain enumerated types treat them as being distinct from the integer types and an explicit cast is required to obtain their numeric value. So the C issues associated with rank do not occur.

---

rank extended integer relative to extended

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank. 668

Commentary

The reasons why an implementation might provide two extended signed integer types of the same precision is the same as the reasons why it might provide such a type having the same precision as a standard integer type. Existing practice provides a ranking for the standard integer types (some or all of which may have the same precision).

664 rank standard integer relative to extended
662 rank standard integer types

C++

The C++ Standard does not specify any properties that must be given to user-defined classes that provide some form of extended integer type.

Coding Guidelines

The same issues apply here as applied to the extended integer types in relation to the standard integer types.

664 rank standard integer relative to extended

---

669 — For all integer types **T1**, **T2**, and **T3**, if **T1** has greater rank than **T2** and **T2** has greater rank than **T3**, then **T1** has greater rank than **T3**.

rank transitive

Commentary

The rank property is transitive.

---

670 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

expression wherever an int may be used

Commentary

An **int** can be thought of as the smallest functional unit of type for arithmetic operations (the types with greater rank being regarded as larger units). This observation is a consequence of the integer promotions. Any integer type can be used in an expression wherever an **int** or **unsigned int** may be used (this may involve them being implicitly converted). However, operands having one of the types specified in the following sentences will often return the same result if they also have the type **int** or **unsigned int**.

675 integer promotions

C90

The C90 Standard listed the types, while the C99 Standard bases the specification on the concept of rank.

*A **char**, a **short int**, or an **int** bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an **int** or **unsigned int** may be used.*

C++

C++ supports the overloading of operators; for instance, a developer-defined definition can be given to the binary **+** operator, when applied to operands having type **short**. Given this functionality, this C sentence cannot be said to universally apply to programs written in C++. It is not listed as a difference because it requires use of C++ functionality for it to be applicable. The implicit conversion sequences are specified in clause 13.3.3.1. When there are no overloaded operators visible (or to be exact no overloaded operators taking arithmetic operands, and no user-defined conversion involving arithmetic types), the behavior is the same as C.

Other Languages

Most other languages do not define integer types that have less precision than type **int**, so they do not contain an equivalent statement. The type **char** is usually a separate type and an explicit conversion is needed if an operand of this type is required in an **int** context.

Coding Guidelines

If the guideline recommendation specifying use of a single integer type is followed, this permission will never be used.

480.1 object int type only
675 integer promotions

Example

In the following:

```
1  #include <limits.h>
2
3  typedef unsigned int T;
4  T x;
5
6  int f(void)
7  {
8  if (sizeof(x) == 2)
9     return (x << CHAR_BIT) << CHAR_BIT;
10 else
11    return sizeof(x);
12 }
```

the first **return** statement will always return zero when the rank of type T is less than or equal to the rank of **int**. There is no guarantee that the second **return** statement will always deliver the same value for different types.

— An object or expression with an integer type whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**. 671

Commentary

rank 663
corresponding signed/unsigned integer promotions 675

The rank of **int** and **unsigned int** is the same. The integer promotions will be applied to these objects.

The wording was changed by the response to DR #230 and allows objects having enumeration type (whose rank may equal the rank of **int** and **unsigned int**) to appear in these contexts (as did C90).

C++

4.5p1 *An rvalue of type* ***char****,* ***signed char****,* ***unsigned char****,* ***short int****, or* ***unsigned short int*** *can be converted to an rvalue of type* ***int*** *if* ***int*** *can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type* ***unsigned int****.*

4.5p2 *An rvalue of type* ***wchar_t*** *(3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type:* ***int****,* ***unsigned int****,* ***long****, or* ***unsigned long****.*

4.5p4 *An rvalue of type* ***bool*** *can be converted to an rvalue of type* ***int****, with* ***false*** *becoming zero and* ***true*** *becoming one.*

The key phrase here is *can be*, which does not imply that they *shall be*. However, the situations where these conversions might not apply (e.g., operator overloading) do not involve constructs that are available in C. For binary operators the *can be* conversions quoted above become *shall be* requirements on the implementation (thus operands with rank less than the rank of **int** are supported in this context):

5p9 *Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:*

— *Otherwise, the integral promotions (4.5) shall be performed on both operands.*[54)]

54) As a consequence, operands of type ***bool****,* ***wchar_t****, or an enumerated type are converted to some integral type.* Footnote 54

The C++ Standard does not appear to contain explicit wording giving this permission for other occurrences of operands (e.g., to unary operators). However, it does not contain wording prohibiting the usage (the wording for the unary operators invariably requires the operand to have an arithmetic or scalar type).

Other Languages

The few languages that do support more than one integer type specify their own rules for when different types can occur in an expression at the same time.

---

672 — A bit-field of type **_Bool**, **int**, **signed int**, or **unsigned int**. bit-field in expression

Commentary

A bit-field is a method of specifying the number of bits to use in the representation of an integer type. The type used in a bit-field declaration specifies the set of possible values that might be available, while the constant value selects the subset (which can include all values) that can be represented by the member. Because the integer promotion rules are based on range of representable values, not underlying signedness of the type, it is possible for a member declared as a bit-field using an unsigned type to be promoted to the type **signed int**. 1393 bit-field maximum width

C90

Support for bit-fields of type **_Bool** is new in C99.

C++

An rvalue for an integral bit-field (9.6) can be converted to an rvalue of type ***int*** *if* ***int*** *can represent all the values of the bit-field; otherwise, it can be converted to* ***unsigned int*** *if* ***unsigned int*** *can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.* 4.5p3

C does not support the definition of bit-fields that are larger than type **int**, or bit-fields having an enumerated type.

Other Languages

Languages, such as Pascal and Ada, provide developers with the ability to specify the minimum and maximum values that need to be represented in an integer type (a bit-field specifies the number of bits in the representation, not the range of values). These languages contain rules that specify how objects defined to have these subrange types can be used anywhere that an object having integer type can appear.

Common Implementations

Obtaining the value of a member that is a bit-field usually involves several instructions. The storage unit holding the bit-field has to be loaded, invariably into a register. Those bits not associated with the bit-field being read then need to be removed. This can involve using a bitwise-and instruction to zero out bits and right shift the bit sequence. For signed bit-fields, it may then be necessary to sign extend the bit sequence. Storing a value into an object having a bit-field type can be even more complex. The new value has to be converted to a bit sequence that fits in the allocate storage, without changing the values of any adjacent objects.

Some CISC processors[968] have instructions designed to access bit-fields. Such relatively complex instructions went out of fashion when RISC design philosophy first took off, but they have started to make a come back.[6,631] Li and Gupta[850] found that adding instructions to the ARM processor that operated (add, subtract, compare, move, and bitwise operations) on subwords reduced the cycle count of various multimedia benchmarks by between 0.39% and 8.67% (code size reductions were between 1.27% and 21.05%).

int can represent values converted to int

If an **int** can represent all values of the original type, the value is converted to an **int**; 673

Commentary

Type conversions occur at translation time, when actual values are usually unknown. The standard requires the translator to assume that the value of the expression can be any one of the representable values supported by its type. While flow analysis could reduce the range of possible values, the standard does not require such analysis to be performed. (If it is performed, a translator cannot use it to change the external behavior of a program; that is, optimizations may be performed but the semantics specified by the standard is followed.)

Other Languages

Most languages have a single signed integer type, so there is rarely a smaller integer type that needs implicit conversion.

Coding Guidelines

typedef assumption of no integer promotions

Some developers incorrectly assume that objects declared using typedef names do not take part in the integer promotions. Incorrect assumptions by a developer are very difficult to deduce from an analysis of the source code. In some cases the misconception will be harmless, the actual program behavior being identical to the misconstrued behavior. In other cases the behavior is different. Guideline recommendations are not a substitute for proper developer training.

Example

```
1  typedef short SHORT;
2
3  extern SHORT es_1,
4              es_2;
5
6  void f(void)
7  {
8  unsigned int ui = 3;    /* Value representable in a signed int.     */
9
10 if (es_1 == (es_2 + 1)) /* Operands converted to int.               */
11    ;
12 if (ui > es_1)          /* Right operand converted to unsigned int. */
13    ;
14 }
```

int cannot represent values converted to unsigned int

otherwise, it is converted to an **unsigned int**. 674

Commentary

This can occur for the types **unsigned short**, or **unsigned char**, if either of them has the same representation as an **unsigned int**. Depending on the type chosen to be compatible with an enumeration type, it is possible for an object that has an enumerated type to be promoted to the type **unsigned int**.

Common Implementations

On 16-bit processors the types **short** and **int** usually have the same representation, so **unsigned short** promotes to **unsigned int**. On 32-bit processors the type **short** usually has less precision than **int**, so the type **unsigned short** promotes to **int**. There are a few implementations, mostly on DSP-based processors, where the character types have the same width as the type **int**.[967]

Coding Guidelines

Existing source code ported, from an environment in which the type **int** has greater width than **short**, to an environment where they both have the same width may have its behavior changed. If the following is executed on a host where the width of type **int** is greater than the width of **short**:

```
1  #include <stdio.h>
2
3  extern unsigned short us;
4  extern signed int si; /* Can hold negative values. */
5
6  void f(void)
7  {
8  if (us > si)
9     printf("Pass\n");
10 else
11    printf("Fail\n");
12 }
```

the object us will be promoted to the type **int**. There will not be any change of values. On a host where the types **int** and **short** have the same width, an **unsigned short** will be promoted to **unsigned int**. This will lead to si being promoted to **unsigned int** (the usual arithmetic conversions) and a potential change in its value. (If it has a small negative value, it will convert to a large positive value.) The relational comparison will then return a different result than in the previous promotion case.

Cg 674.1

An object having an unsigned integer type shall not be implicitly converted to **unsigned int** through the application of the integer promotions.

The consequence of this guideline recommendation is that such conversions need to be made explicit, using a cast to an integer type whose rank is greater than or equal to **int**.

---

675 These are called the *integer promotions*.[48)]

integer promotions

Commentary

This defines the term *integer promotions*. Integer promotions occur when an object having a rank less than **int** appears in certain contexts. This behavior differs from arithmetic conversions where the type of a different object is involved. Integer promotions are affected by the relative widths of types (compared to the width of **int**). If the type **int** has greater width than **short** then, in general (the presence of extended integer types whose rank is also less than **int** can complicate the situation), all types of less rank will convert to **int**. If **short** has the same precision as **int**, an **unsigned short** will invariably promote to an **unsigned int**.

690 footnote 48

It is possible to design implementations where the integer conversions don't follow a simple pattern, such as the following:

```
signed short 16 bits including sign     unsigned short 24 bits
signed int   24 bits including sign     unsigned int   32 bits
```

Your author does not know of any implementation that uses this kind of unusual combination of bits for its integer type representation.

C90

These are called the integral promotions.[27)]

C++

The C++ Standard uses the C90 Standard terminology (and also points out, 3.9.1p7, "A synonym for integral type is *integer type*.").

Other Languages

The *unary numeric promotions* and *binary numeric promotions* in Java have the same effect.

Common Implementations

Many processors have load instructions that convert values having narrower types to a wider type. For instance, loading a byte into a register and either sign extending (**signed char**), or zero filling (**unsigned char**) the value to occupy 32 bits (promotion to **int**). On processors having instructions that operate on values having a type narrower than **int** more efficiently than type **int**, optimizers can make use of the as-if rule to improve efficiency. For instance, in some cases an analysis of the behavior of a program may find that operand values and the result value is always representable in the their unpromoted type. Implementations need only to act as if the object had been converted to the type **int**, or **unsigned int**.

Coding Guidelines

object 480.1 int type only

operand 653 convert automatically

If the guideline recommendation specifying use of a single integer type is followed there would never be any integer promotions. The issue of implicit conversions versus explicit conversions might be a possible cause of a deviation from this recommendation and is discussed elsewhere.

Example

```
1  signed short    s1,  s2,  s3;
2  unsigned short us1, us2, us3;
3
4  void f(void)
5  {
6  s1 = s2 + s3;  /*
7                  * The result of + may be undefined.
8                  * The conversion for the = may be undefined.
9                  */
10                /* s1 = (short)((int)s2 + (int)s3);           */
11 s1 = us2 + s3; /* The conversion for the = may be undefined. */
12                /*
13                 * The result of the binary + is always defined (unless
14                 * the type int is only one bit wider than a short; no
15                 * known implementations have this property).
16                 *
17                 * Either both shorts promote to a wider type:
18                 *
19                 *       s1 = (short)((int)us2 + (int)s3);
20                 *
21                 * or they both promote to an unsigned type of the same width:
22                 *
23                 *       s1 = (short)((unsigned int)us2 + (unsigned int)s3);
24                 */
25  s1 = us2 + us3; /* The conversion for the = may be undefined. */
26 us1 = us2 + us3; /* Always defined */
27 us1 = us2 +  s3; /* Always defined */
28 us1 =  s2 +  s3; /* The result of + may undefined.             */
29 }
```

Table 675.1: Occurrence of integer promotions (as a percentage of all operands appearing in all expressions). Based on the translated form of this book's benchmark programs.

Original Type	%	Original Type	%
unsigned char	2.3	**char**	1.2
unsigned short	1.9	**short**	0.5

All other types are unchanged by the integer promotions.

Commentary

The integer promotions are only applied to values whose integer type has a rank less than that of the **int** type.

C++

This is not explicitly specified in the C++ Standard. However, clause 4.5, Integral promotions, discusses no other types, so the statement is also true in C++

---

677 The integer promotions preserve value including sign. value preserving

Commentary

These rules are sometimes known as *value preserving promotions*. They were chosen by the Committee because they result in the least number of surprises to developers when applied to operands. The promoted value would remain unchanged whichever of the two rules used by implementations were used. However, in many cases this promoted value appears as an operand of a binary operator. If *unsigned preserving promotions* were used (see Common implementations below), the value of the operand could have its sign changed (e.g., if the operands had types **unsigned char** and **signed char**, both their final operand type would have been **unsigned int**), potentially leading to a change of that value (if it was negative). The *unsigned preserving promotions* (sometimes called rules rather than promotions) are sometimes also known as *sign preserving rules* because the form of the sign is preserved.

Most developers think in terms of values, not signedness. A rule that attempts to preserve sign can cause a change of value, something that is likely to be unexpected. Value preserving rules can also produce results that are unexpected, but these occur much less often.

Rationale

The *unsigned preserving* rules greatly increase the number of situations where **unsigned int** confronts **signed int** to yield a questionably signed result, whereas the *value preserving* rules minimize such confrontations. Thus, the value preserving rules were considered to be safer for the novice, or unwary, programmer. After much discussion, the C89 Committee decided in favor of value preserving rules, despite the fact that the UNIX C compilers had evolved in the direction of unsigned preserving.

Other Languages

This is only an issue for languages that contain more than one signed integer type and an unsigned integer type.

Common Implementations

The base document specified unsigned preserving rules. If the type being promoted was either **unsigned char** or **unsigned short**, it was converted to an **unsigned int**. The corresponding signed types were promoted to **signed int**. Some implementations provide an option to change their default behavior to follow unsigned preserving rules.[600, 1314, 1340] [1] base document

Coding Guidelines

Existing, very old, source code may rely on using the unsigned preserving rules. It can only do this if the translator is also running in such a mode, either because that is the only one available or because the translator is running in a compatibility mode to save on the porting (to the ISO rules) cost. Making developers aware of any of the issues involved in operating in a nonstandard C environment is outside the scope of these coding guidelines.

Example

```
1  extern unsigned char uc;
2
3  void f(void)
```

```
4   {
5   int si = -1;
6   /*
7    * Value preserving rules promote uc to an int -> comparison succeeds.
8    *
9    * Signed preserving rules promote uc to an unsigned int, usual arithmetic
10   * conversions then convert si to unsigned int -> comparison fails.
11   */
12  if (uc > si)
13     ;
14  }
```

---

char plain treated as

As discussed earlier, whether a "plain" **char** is treated as signed is implementation-defined. 678

Commentary

char 516 range, representation and behavior

The implementation-defined treatment of "plain" char will only affect the result of the integer promotions if any of the character types can represent the same range of values as an object of type **int** or **unsigned int**.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1). 679

6.3.1.2 Boolean type

_Bool converted to

When any scalar value is converted to **_Bool**, the result is 0 if the value compares equal to 0; 680

Commentary

Converting a scalar value to type **_Bool** is effectively the same as a comparison against 0; that is, `(_Bool)x` is effectively the same as `(x != 0)` except in the latter case the type of the result is **int**.

Conversion to **_Bool** is different from other conversions, appearing in a strictly conforming program, in that it is not commutative— `(T1)(_Bool)x` need not equal `(_Bool)(T1)x`. For instance:

```
(int)(_Bool)0.5 ⇒ 1
(_Bool)(int)0.5 ⇒ 0
```

Reordering the conversions in a conforming program could also return different results:

```
(signed)(unsigned)-1 ⇒ implementation-defined
(unsigned)(signed)-1 ⇒ UINT_MAX
```

C90

Support for the type **_Bool** is new in C99.

C++

4.12p1 *An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type* ***bool****. A zero value, null pointer value, or null member pointer value is converted to* ***false****;*

The value of **false** is not defined by the C++ Standard (unlike **true**, it is unlikely to be represented using any value other than zero). But in contexts where the integer conversions are applied:

4.7p4 *. . . the value* ***false*** *is converted to zero . . .*

Other Languages

Many languages that include a boolean type specify that it can hold the values true and false, without specifying any representation for those values. Java only allows boolean types to be converted to boolean types. It does not support the conversion of any other type to boolean.

Coding Guidelines

The issue of treating boolean values as having a well-defined role independent of any numeric value is discussed elsewhere; for instance, treating conversions of values to the type **_Bool** as representing a change of role, not as representing the values 0 and 1. The issue of whether casting a value to the type **_Bool**, rather than comparing it against zero, represents an idiom that will be recognizable to C developers is discussed elsewhere.

476 boolean role

476 boolean role

---

681 otherwise, the result is 1.

Commentary

In some contexts C treats any nonzero value as representing true — for instance, controlling expressions (which are also defined in terms of a comparison against zero). A conversion to **_Bool** reduces all nonzero values to the value 1.

1744 if statement operand compare against 0

C++

> *. . . ; any other value is converted to* ***true****.* 4.12p1

The value of **true** is not defined by the C++ Standard (implementations may choose to represent it internally using any nonzero value). But in contexts where the integer conversions are applied:

> *. . . the value* ***true*** *is converted to one.* 4.7p4

6.3.1.3 Signed and unsigned integers

---

682 When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged.

Commentary

While it would very surprising to developers if the value was changed, the standard needs to be complete and specify the behavior of all conversions. For integer types this means that the value has to be within the range specified by the corresponding numerical limits macros.

300 numerical limits

The type of a bit-field is more than just the integer type used in its declaration. The width is also considered to be part of its type. This means that assignment, for instance, to a bit-field object may result in the value being assigned having its value changed.

1407 bit-field interpreted as

```
1   void DR_120(void)
2   {
3   struct {
4            unsigned int mem : 1;
5           } x;
6   /*
7    * The value 3 can be represented in an unsigned int,
8    * but is changed by the assignment in this case.
9    */
10  x.mem = 3;
11  }
```

C90

Support for the type **_Bool** is new in C99, and the C90 Standard did not need to include it as an exception.

Other Languages

This general statement holds true for conversions in other languages.

Common Implementations

The value being in range is not usually relevant because most implementations do not perform any range checks on the value being converted. When converting to a type of lesser rank, the common implementation behavior is to ignore any bit values that are not significant in the destination type. (The sequence of bits in the value representation of the original type is truncated to the number of bits in the value representation of the destination type.) If the representation of a value does not have any bits set in these ignored bits, the converted value will be the same as the original value. In the case of conversions to value representations containing more bits, implementations simply sign-extend for signed values and zero-fill for unsigned values.

Coding Guidelines

One way of reducing the possibility that converted values are not representable in the converted type is to reduce the number of conversions. This is one of the rationales behind the general guideline on using a single integer type.

object 480.1 int type only

---

unsigned integer conversion to

Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[49)] 683

Commentary

This behavior is what all known implementations do for operations on values having unsigned types. The standard is enshrining existing processor implementation practices in the language. As footnote 49 points out, this adding and subtracting is done on the abstract mathematical value, not on a value with a given C type. There is no need to think in terms of values wrapping (although this is a common way developers think about the process).

footnote 691 49

C90

Otherwise: if the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.[28)]

When a value with integral type is demoted to an unsigned integer with smaller size, the result is the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size.

The C99 wording is a simpler way of specifying the C90 behavior.

Common Implementations

For unsigned values and signed values represented using two's complement, the above algorithm can be implemented by simply chopping off the significant bits that are not available in the representation of the new type.

Coding Guidelines

The behavior for this conversion may be fully specified by the standard. The question is whether a conversion should be occurring in the first place.

unsigned 496 computation modulo reduced

---

integer value not represented in signed integer

Otherwise, the new type is signed and the value cannot be represented in it; 684

Commentary

To be exact, the standard defines no algorithm for reducing the value to make it representable (because there is no universal agreement between different processors on what to do in this case).

Other Languages

The problem of what to do with a value that, when converted to a signed integer type, cannot be represented is universal to all languages supporting more than one signed integer type, or support an unsigned integer type (the overflow that can occur during an arithmetic operation is a different case).

Coding Guidelines

A guideline recommendation that the converted value always be representable might be thought to be equivalent to one requiring that a program not contain defects. However, while the standard may not specify an algorithm for this conversion, there is a commonly seen implementation behavior. Developers sometimes intentionally make use of this common behavior and the applicable guideline is the one dealing with the use of representation information.

569.1 representation information using

---

685 either the result is implementation-defined or an implementation-defined signal is raised.

signed integer conversion implementation-defined

Commentary

There is no universally agreed-on behavior (mathematical or processor) for the conversion of out-of-range signed values, so the C Standard's Committee could not simply define this behavior as being what happens in practice. The definition of implementation-defined behavior does not permit an implementation to raise a signal; hence, the additional permission to raise a signal is specified here.

42 implementation-defined behavior

C90

The specification in the C90 Standard did not explicitly specify that a signal might be raised. This is because the C90 definition of implementation-defined behavior did not rule out the possibility of an implementation raising a signal. The C99 wording does not permit this possibility, hence the additional permission given here.

C++

. . . ; otherwise, the value is implementation-defined. 4.7p3

The C++ Standard follows the wording in C90 and does not explicitly permit a signal from being raised in this context because this behavior is considered to be within the permissible range of implementation-defined behaviors.

Other Languages

Languages vary in how they classify the behavior of a value not being representable in the destination type. Java specifies that all the unavailable significant bits (in the destination type) are discarded. Ada requires that an exception be raised. Other languages tend to fall between these two extremes.

Common Implementations

The quest for performance and simplicity means that few translators generate machine code to check for nonrepresentable conversions. The usual behavior is for the appropriate number of least significant bits from the original value to be treated as the converted value. The most significant bit of this new value is treated as a sign bit, which is sign-extended to fill the available space if the value is being held in a register. If the conversion occurs immediately before a store (i.e., a right-hand side value is converted before being assigned into the left hand side object), there is often no conversion; the appropriate number of value bits are simply written into storage.

Some older processors[284] have the ability to raise a signal if a conversion operation on an integer value is not representable. On such processors an implementation can choose to use this instruction or alternatively use a sequence of instructions having the same effect, that do not raise a signal.

6.3.1.4 Real floating and integer

floating-point converted to integer

When a finite value of real floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). 686

Commentary

NaNs are not finite values and neither are they infinities.

IEEE-754 6.3 *The Sign Bit. . . . and the sign of the result of the round floating-point number to integer operation is the sign of the operand. These rules shall apply even when operands or results are zero or infinite.*

negative zero 616 only generated by

When a floating-point value in the range (-1.0, -0.0) is converted to an integer type, the result is required to be a positive zero.

C90

Support for the type **_Bool** is new in C99.

Other Languages

This behavior is common to most languages.

Common Implementations

FLT_ROUNDS 352

Many processors include instructions that perform truncation when converting values of floating type to an integer type. On some processors the rounding mode, which is usually set to round-to-nearest, has to be changed to round-to-zero for this conversion, and then changed back after the operation. This is an execution-time overhead. Some implementations give developers the choice of faster execution provided they are willing to accept round-to-nearest behavior. In some applications the difference in behavior is significantly less than the error in the calculation, so it is acceptable.

Coding Guidelines

integer constant expression 1328

An expression consisting of a cast of a floating constant to an integer type is an integer constant expression. Such a constant can be evaluated at translation time. However, there is no requirement that the translation-time evaluation produce exactly the same results as the execution-time evaluation. Neither is there a requirement that the translation-time handling of floating-point constants be identical. In the following example it is possible that a call to `printf` will occur.

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  float fval = 123456789.0;
6  long lval = (long)123456789.0;
7
8  if (lval != (long)fval)
9     printf("(long)123456789.0 == %ld and %ld\n", lval, (long)fval);
10 }
```

There is a common, incorrect, developer assumption that floating constants whose fractional part is zero are always represented exactly by implementations (i.e., many developers have a mental model that such constants are really integers with the characters `.0` appended to them). While it is technically possible to convert many such constants exactly, experience shows that a surprising number of translators fail to achieve the required degree of accuracy (e.g., the floating constant 6.0 might be translated to the same internal representation as the floating constant 5.999999 and subsequently converted to the integer constant 5).

Rev 686.1 A program shall not depend on the value of a floating constant being converted to an integer constant having the same value.

A developer who has made the effort of typing a floating constant is probably expecting it to be used as a floating type. Based on this assumption a floating constant that is implicitly converted to an integer type is unexpected behavior. Such an implicit conversion can occur if the floating constant is the right operand of an assignment or the argument in a function call. Not only is the implicit conversion likely to be unexpected by the original author, but subsequent changes to the code that cause a function-like macro to be invoked, rather than a function call, to result in a significant change in behavior.

In the following example, a floating constant passed to `CALC_1` results in `glob` being converted to a floating type. If the value of `glob` contains more significant digits than supported by the floating type, the final result assigned to `loc` will not be the value expected. Using explicit casts, as in `CALC_2`, removes the problem caused by the macro argument having a floating type. However, as discussed elsewhere, other dependencies are introduced. Explicitly performing the cast, where the argument is passed, mimics the behavior of a function call and shows that the developer is aware of the type of the argument.

653 operand convert automatically

```
#define X_CONSTANT 123456789.0
#define Y_CONSTANT 2

#define CALC_1(a)  ((a) + (glob))
#define CALC_2(a)  ((long)(a) + (glob))
#define CALC_3(a)  ((a) + (glob))

extern long glob;

void f(void)
{
long loc;

loc = CALC_1(X_CONSTANT);
loc = CALC_1(Y_CONSTANT);

loc = CALC_2(X_CONSTANT);
loc = CALC_2(Y_CONSTANT);

loc = CALC_3((long)X_CONSTANT);
loc = CALC_3(Y_CONSTANT);
}
```

The previous discussion describes some of the unexpected behaviors that can occur when a floating constant is implicitly converted to an integer type. Some of the points raised also apply to objects having a floating type. The costs and benefits of relying on implicit conversions or using explicit casts are discussed, in general, elsewhere. That discussion did not reach a conclusion that resulted in a guideline recommendation being made. Literals differ from objects in that they are a single instance of a single value. As such developers have greater control over their use, on a case by case basis, and a guideline recommendation is considered to be more worthwhile. This guideline recommendation is similar to the one given for conversions of suffixed integer constants.

653 operand convert automatically

835.2 integer constant with suffix, not immediately converted

Cg 686.2 A floating constant shall not be implicitly converted to an integer type.

687 If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[50)]

Commentary

The exponent part in a floating-point representation allows very large values to be created, these could significantly exceed the representable range supported by any integer type. The behavior specified by the standard reflects both the fact that there is no commonly seen processor behavior in this case and the execution-time overhead of performing some defined behavior.

Other Languages

Other languages vary in their definition of behavior. Like integer values that are not representable in the destination type, some languages require an exception to be raise while others specify undefined behavior. In this case Java uses a two step-process. It first converts the real value to the most negative, or largest positive (depending on the sign of the floating-point number) value representable in a **long** or an **int**. In the second step, if the converted integer type is not **long** or **int**; the narrowing conversions are applied to the result of the first step.

Common Implementations

Many processors have the option of raising an exception when the value cannot be represented in the integer type. Some allow these exceptions to be switched off, returning the least significant bytes of the value. The IEC 60559 Standard defines the behavior— raise invalid. Most current C99 implementations do not do this.

arithmetic saturated

In graphics applications saturated arithmetic is often required (see Figure 687.1). Some DSP processors[967] and the Intel MMX instructions[627] return the largest representable value (with the appropriate sign).

Coding Guidelines

A naive analysis would suggest there is a high probability that an object having a floating type will hold a value that cannot be represented in an integer type. However, in many programs the range of floating-point values actually used is relatively small. It is this application-specific knowledge that needs to be taken into account by developers.

exponent 334

Those translators that perform some kind of flow analysis on object values often limit themselves to tracking the values of integer and pointer types. Because of the potential graininess in the values they represent and their less common usage, objects having floating types may have their set/unset status tracked but their possible numeric value is rarely tracked.

It might appear that, in many ways, this case is the same as that for integer conversions where the value cannot be represented. However, a major difference is processor behavior. There is greater execution overhead required for translators to handle this case independently of how the existing instructions behave. Also, a larger number of processors are capable of raising an exception in this case.

integer value 684
not represented in signed integer

Given that instances of this undefined behavior are relatively rare and instances might be considered to be a fault, no guideline recommendation is made here.

---

integer conversion to floating

When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. 688

Commentary

The value may be unchanged, but its representation is likely to be completely changed.

There are the same number of representable floating-point values between every power of two (when `FLT_RADIX` has a value of two, the most common case). As the power of two increases, the numeric distance between representable values increases (see Figure 368.1). The value of the `*_DIG` macros specify the number of digits in a decimal value that may be rounded into a floating-point number and back again without

FLT_RADIX 366
*_DIG 369
macros

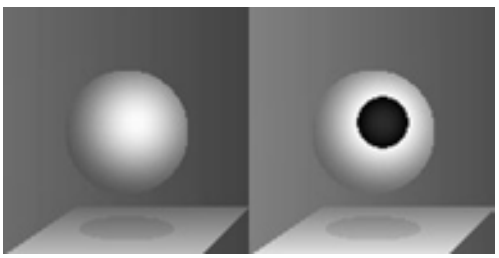


Figure 687.1: Illustration of the effect of integer addition wrapping rather than saturating. A value has been added to all of the pixels in the left image to increase the brightness, creating the image on the right. With permission from Jordán and Lotufo.[692]

change of value. In the case of the single-precision IEC 60559 representation FLT_DIG is six, which is less than the number of representable digits in an object having type **long** (or 32-bit **int**).

C++

An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible. 4.9p2

Who decides what is possible or if it can be represented exactly? A friendly reading suggests that the meaning is the same as C99.

Common Implementations

The Unisys A Series[1389] uses the same representation for integer and floating-point types (an integer was a single precision value whose exponent was zero).

Coding Guidelines

A common, incorrect belief held by developers is that because floating numbers can represent a much larger range of values than integers and include information on fractional parts, they must be able to exactly represent the complete range of values supported by any integer type. What is overlooked is that the support for an exponent value comes at the cost of graininess in the representation for large value (if objects of integer and floating type are represented in the same number of value bits).

The major conceptual difference between integer and floating types is that one is expected to hold an exact value and the other an approximate value. If developers are aware that approximations can begin at the point an integer value is converted, then it is possible for them to take this into account in designing algorithms. Developers who assume that inaccuracies don't occur until the floating value is operated on are in for a surprise.

Rev 688.1 Algorithms containing integer values that are converted to floating values shall be checked to ensure that any dependence on the accuracy of the conversion is documented and that any necessary execution-time checks against the *_DIG macros are made.

The rationale behind the guideline recommendations against converting floating constants to integer constants do not apply to conversions of integer constants to floating types. 686.2 floating constant implicitly converted

Example

```
#include <limits.h>
#include <stdio.h>

static float max_short = (float)SHRT_MAX;
static float max_int = (float)INT_MAX;
static float max_long = (float)LONG_MAX;

int main(void)
{
float max_short_m1,
      max_int_m1,
      max_long_m1;

for (int i_f=1; i_f < 3; i_f++)
   {
   max_short_m1 = (float)(SHRT_MAX - i_f);
   if (max_short == max_short_m1)
      printf("float cannot represent all representable shorts\n");
```

```
19      max_short=max_short_m1;
20      max_int_m1 = (float)(INT_MAX - i_f);
21      if (max_int == max_int_m1)
22         printf("float cannot represent all representable ints\n");
23      max_int=max_int_m1;
24      max_long_m1 = (float)(LONG_MAX - i_f);
25      if (max_long == max_long_m1)
26         printf("float cannot represent all representable longs\n");
27      max_long=max_long_m1;
28      }
29   }
```

---

int to float nearest representable value

If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. 689

Commentary

There is no generally accepted behavior in this situation. The standard leaves it up to the implementation.

Other Languages

Most languages are silent on this issue.

Common Implementations

status flag 200 floating-point FLT_ROUNDS 352

Most processors contain a status flag that is used to control the rounding of all floating-point operations. Given that round-to-nearest is the most common rounding mode, the most likely implementation-defined behavior is round-to-nearest.

Coding Guidelines

A consequence of this behavior is that it is possible for two unequal integer values to be converted to the same floating-point value. Any algorithm that depends on the relationships between integer values being maintained after conversion to a floating type may not work as expected.

---

footnote 48

48) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, -, and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses. 690

Commentary

The certain argument expressions are function calls where there is no type information available. This happens for old-style function declarations and for prototype declarations where the ellipsis notation has been used and the argument corresponds to one of the parameters covered by this notation.

Another context where integer promotions are applied is the controlling expression in a **switch** statement. In all other cases the operand is being used in a context where its value is being operated on.

Contexts where the integer promotions are not applied are the expression specifying the number of elements in a VLA type definition (when there are no arithmetic or logical operators involved), function return values, the operands of the assignment operator, and arguments to a function where the parameter type is known. In the latter three cases the value is, potentially, implicitly cast directly to the destination type.

if statement 1744 operand compare against 0 iteration statement 1766 executed repeatedly

The standard does not specify that the implicit test against zero, in an **if** statement or iteration statement, will cause a single object (forming the complete controlling expression) to be promoted. A promotion would not affect the outcome in these contexts, and an implementation can use the as-if rule in selecting the best machine code to generate.

C++

In C++, integral promotions are applied also as part of the usual arithmetic conversions, the operands of the unary +, -, and ~ operators, and to both operands of the shift operators. C++ also performs integer promotions in contexts not mentioned here, as does C.

Coding Guidelines

There are a small number of cases where the integer promotions do not occur. Is anything to be gained by calling out these situations in coding guideline documents? Experience suggests that developers are more likely to forget that the integer promotions occur (or invent mythical special cases where they don't occur) rather than worry about additional conversions because of them. Coding guideline documents are no substitute for proper training.

---

691 49) The rules describe arithmetic on the mathematical value, not the value of a given type of expression. footnote 49

Commentary

This means that there are no representational issues involving intermediate results being within range of the type of the values. The abstract machine must act as if the operation were performed using infinite precision arithmetic.

C90

This observation was not made in the C90 Standard (but was deemed to be implicitly true).

C++

The C++ Standard does not make this observation.

---

692 50) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. footnote 50

Commentary

This permission reflects both differences in processor instruction behavior and the (in)ability to detect and catch those cases where the conversion might be performed in software. The behavior is essentially unspecified, although it is not explicitly specified as such (and it does not appear in the list of unspecified behaviors in Annex J.1).

C++

An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. 4.9p1

The conversion behavior, when the result cannot be represented in the destination type is undefined in C++ and unspecified in C.

Common Implementations

The floating-point to integer conversion instructions on many processors are only capable of delivering signed integer results. An implementation may treat the value as a sequence of bits independent of whether a signed or unsigned value is expected. In this case the external behavior for two's complement notation is the same as if a remaindering operation had been performed. Converting values outside of the representable range of any integer type supported by an implementation requires that the processor conversion instruction either perform the remainder operation or raise some kind of range error signal that is caught by the implementation, which then performs the remainder operation in software.

Coding Guidelines

This is one area where developers' expectations may not mirror the behavior that an implementation is required to support.

Example

The following program may, or may not, output the values given in the comments.

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
double d = UINT_MAX;
printf("%f, %u\n", d, (unsigned int)d); /* 4294967295.000000, 4294967295 */
d += 42;
printf("%f, %u\n", d, (unsigned int)d); /* 4294967337.000000, 41         */
d *= 20;
printf("%f, %u\n", d, (unsigned int)d); /* 85899346740.000000, 820       */
d = -1;
printf("%f, %u\n", d, (unsigned int)d); /* -1.000000, 4294967295         */
}
```

Thus, the range of portable real floating values is $(-1, \textit{Utype\_}\texttt{MAX} + 1)$. 693

Commentary

The round brackets are being used in the mathematical sense; the bounds represent excluded limits (i.e., -1 is not in the portable range). This statement only applies to unsigned integer types.

C++

The C++ Standard does not make this observation.

If the value being converted is outside the range of values that can be represented, the behavior is undefined. 694

Commentary

*_MAX_10_EXP[373]

An unsigned integer type represented in 123 bits, or more, could contain a value that would be outside the range of values representable in a minimum requirements **`float`** type (128 bits would be needed to exceed the range of the IEC 60559 single-precision).

C++

4.9p2 *Otherwise, it is an implementation-defined choice of either the next lower or higher representable value.*

The conversion behavior, when the result is outside the range of values that can be represented in the destination type, is implementation-defined in C++ and undefined in C.

Other Languages

Many other language standards were written in an age when floating-point types could always represent much larger values than could be represented in integer types and their specifications reflect this fact (by not mentioning this case). In Java values having integer type are always within the representable range of the floating-point types it defines.

Common Implementations

Processors that support 128-bit integer types, in hardware, are starting to appear.[27]

Coding Guidelines

Developers are likely to consider this issue to be similar to the year 2036 problem— address the issue when the course of history requires it to be addressed.

6.3.1.5 Real floating types

695 When a **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double**, its value is unchanged (if the source value is represented in the precision and range of its type). float promoted to double or long double

Commentary

Although not worded as such, this is a requirement on the implementation. The type **double** must have at least the same precision in the significand and at least as much range in the exponent as the type **float**, similarly for the types **double** and **long double**.

A situation where the source value might not be represented in the precision and range of its type is when FLT_EVAL_METHOD has a non-zero value. For instance, if FLT_EVAL_METHOD has a value of 2, then the value `0.1f` is represented to the precision of **long double**, while the type remains as **float**. If a cast to **double** is performed<footnote, 87> the value may be different to that obtained when FLT_EVAL_METHOD was zero. [354] FLT_EVAL_ME

The wording was changed by the response to DR #318.

C++

*The type **double** provides at least as much precision as **float**, and the type **long double** provides at least as much precision as **double**. The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.* 3.9.1p8

This only gives a relative ordering on the available precision. It does not say anything about promotion leaving a value unchanged.

*An rvalue of type **float** can be converted to an rvalue of type **double**. The value is unchanged.* 4.6p1

There is no equivalent statement for type **double** to **long double** promotions. But there is a general statement about conversion of floating-point values:

An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. 4.8p1

Given that (1) the set of values representable in a floating-point type is a subset of those supported by a wider floating-point type (3.9.1p8); and (2) when a value is converted to the wider type, the exact representation is required to be used (by 4.8p1)— the value must be unchanged.

Other Languages

Some languages specify one floating type, while others recognize that processors often support several different floating-point precisions and define mechanisms to allow developers to specify different floating types. While implementations that provide multiple floating-point types usually make use of the same processor hardware as that available to a C translator, other languages rarely contain this requirement.

Common Implementations

This requirement is supported by IEC 60559, where each representation is an extension of the previous ones holding less precision.

696 When a **double** is demoted to **float**, a **long double** is demoted to **double** or **float**, or a value being represented in greater precision and range than required by its semantic type (see 6.3.1.8) is explicitly converted <iso_delete>to its semantic type</iso_delete> (including to its own type), if the value being converted can be represented exactly in the new type, it is unchanged. double demoted to another floating type

Commentary

A simple calculation would suggest that unless an implementation uses the same representation for floating-point types, the statistically likelihood of a demoted value being exactly representable in the new type would be very low (an IEC 60559 double-precision type contains 2^{32} values that convert to the same single-precision value). However, measurements of floating-point values created during program execution show a surprisingly high percentage of value reuse (these results are discussed elsewhere).

binary * 1147 result
value 940
profiling
354 FLT_EVAL_METHOD
float 695 promoted to double or long double

A situation where a value might be represented in greater precision and range than required by its type is when FLT_EVAL_METHOD has a non-zero value.

The wording was changed by the response to DR #318.

C90

This case is not specified in the C90 Standard.

Coding Guidelines

equality 1214.1 operators not floating-point operands

Relying on the converted value being equal to the original value is simply a special case of comparing two floating-point values for equality.

Why is a floating-point value being demoted? If a developer is concerned about a floating-point value being represented to a greater precision than required by its semantic type, explicit conversions might be used simply as a way of ensuring known behavior in the steps in a calculation. Or perhaps a computed value is being assigned to an object. There are sometimes advantages to carrying out the intermediate steps in an expression evaluation in greater precision than the result that will be saved.

354 FLT_EVAL_METHOD
correctly 64 rounded result

If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. 697

Commentary

352 FLT_ROUNDS

Although FLT_ROUNDS is only defined to characterize floating-point addition, its value is very likely to characterize the behavior in this case as well.

Other Languages

IEC 60559 29

Java specifies the IEEE-754 round-to-nearest mode.

Common Implementations

352 FLT_ROUNDS

Many implementations round-to-nearest. However, rounding to zero (ignoring the least significant bits) can have performance advantages. Changing the rounding mode of the hardware floating-point unit requires an instruction to be executed, which in itself is a fast instruction. However, but a common side effect is to require that the floating-point pipeline be flushed before another floating-point instruction can be issued— an expensive operation. Leaving the processor in round-to-zero mode may offer savings. The IBM S/360, when using its hexadecimal floating point representation, truncates a value when converting it to a representation having less precision.

Coding Guidelines

Like other operations on floating values, conversion can introduce a degree of uncertainty into the result. This is a characteristic of floating point; there are no specific guidelines for this situation.

---

floating value converted not representable

If the value being converted is outside the range of values that can be represented, the behavior is undefined. 698

Commentary

For very small values there is always a higher and lower value that bound them. The situation describes in the C sentence can only occur if the type being demoted from is capable of representing a greater range of exponent values than the destination type.

Other Languages

A value too small to be represented as a **`float`** *is converted to positive or negative zero; a value too large to be represented as a* **`float`** *is converted to a (positive or negative) infinity.* In Java

Common Implementations

On many processors the result of the conversion is controlled by the rounding mode. A common behavior is to return the largest representable value (with the appropriate sign) if rounding to zero is in force, and to infinity (with the appropriate sign) if rounding to nearest is in force. When rounding to positive or negative infinity, the result is either the largest representable value or infinity, depending on the sign of the result and the sign of the infinity being rounded to. Many processors also have the ability to raise some form of overflow exception in this situation, which can often be masked. [352] FLT_ROUNDS

In signal processing applications infinity is not a sensible value to round to, and processors used for these kinds of applications often saturate at the largest representable value.

Coding Guidelines

The issue of a value not being representable, in the destination type, applies to many floating-point operations other than conversion.

A result of infinity is the required behavior in some applications because it has known properties and its' effect on subsequent calculations might be intended to produce a result of infinity, or zero (for divide operations). Tests can be inserted into the source code to check for infinite results. In other applications, typically graphics or signal processing-oriented ones, a saturated value is required and developers would not expect to need to check for overflow.

Performance is often an important issue in code performing floating-point operations (adding code to do range checks can cause excessive performance penalties). Given the performance issue and the variety of possible application-desired behaviors and actual processor behaviors, there is no obvious guideline recommendation that can be made.

6.3.1.6 Complex types

699 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

Commentary

A complex type is the sum of its two parts in the Cartesian system. There is no requirement to minimize the difference between the modulus of the converted value and the modulus of the original value.

C90

Support for complex types is new in C99.

C++

The C++ Standard does not provide a specification for how the conversions are to be implemented.

Other Languages

The Fortran intrinsic function, `CMPLX` (whose behavior is mimicked on assignment), can take a complex parameter that specifies the type of conversion that should occur. If no `KIND` is specified, the intrinsic takes the value applicable to the default real type.

6.3.1.7 Real and complex

700 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero. real type converted to complex

Commentary

Prior to the conversion, the value has an arithmetic type and is in the real type domain. After the conversion the value has an arithmetic type and is in the complex type domain.

In mathematical terms all real numbers are part of the complex plane, with a zero imaginary part. It is only when a real type is converted to a complex type that a C implementation needs to start to associate an imaginary value with the complete value.

negative zero 615

The imaginary part is never a negative zero.

C90

Support for complex types is new in C99.

C++

The constructors for the complex specializations, 26.2.3, take two parameters, corresponding to the real and imaginary part, of the matching floating-point type. The default value for the imaginary part is specified as 0.0.

Other Languages

The Fortran intrinsic function `CMPLX` takes a parameter that specifies the type of conversion that should occur. If no `KIND` is specified, the intrinsic takes the value of the default real type. This intrinsic function also provides a default imaginary value of $0i$ (but does not say anything about the representation of zero).

Coding Guidelines

Support for complex types is new in C99 and there is no experience based on existing usage to draw on. Are there any parallels that can be made with other constructs (with a view to adapting guidelines that have been found to be useful for integer constants)? Some parallels are discussed next; however, no guideline recommendation is made because usage of complex types is not sufficiently great for there to be a likely worthwhile benefit.

Conversions between integer types and real floating types, and conversions between real floating to complex floating, both involve significant changes in developer conception and an implementation's internal representation. However, these parallels do not appear to suggest any worthwhile guideline recommendation.

A value of real type can also be converted to a complex type by adding an imaginary value of zero to it— for instance, `+0.0I` (assuming an implementation supports this form of constant). However, casts are strongly associated with conversion in developers' minds. The binary `+` operation may cause conversions, but this is not its primary association. The possibility that an implementation will not support this literal form of imaginary values might be considered in itself sufficient reason to prefer the use of casts, even without the developer associations.

```
1  #include <complex.h>
2
3  extern double _Complex glob;
4
5  void f(void)
6  {
7  glob = 1.0;                    /* Implicit conversion.      */
8  glob = (double _Complex)1.0;   /* Explicit conversion.      */
9  glob = 1.0 + 0.0I;             /* I might not be supported. */
10 }
```

Conversion of a value having a complex type, whose two components are both constants (the standard does not define the term *complex constant*), to a non-complex type is suspicious for the same reason as are other conversions of constant values.

floating-point 686
converted to integer

Are there any issues specifically associated with conversion to or from complex types that do not apply for conversions between other types? It is a change of type domain. At the time if writing there is insufficient experience available, with this new type, to know whether these issues are significant.

701 When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

Commentary

This conversion simply extracts the real part from a complex value. It has the same effect as a call to the `creal` library function. A NaN value in the part being discarded does not affect the value of the result (rather than making the result value a NaN). There are no implicit conversions defined for converting to the type **`_Imaginary`**. The library function `cimag` has to be called explicitly.

1378 type specifier syntax

C++

In C++ the conversion has to be explicit. The member functions of the complex specializations (26.2.3) return a value that has the matching floating-point type.

Other Languages

Some languages support implicit conversions while others require an explicit call to a conversion function.

6.3.1.8 Usual arithmetic conversions

702 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way.

operators cause conversions

Commentary

The operators cause the conversions in the sense that the standard specifies that they do. There is no intrinsic reason why they have to occur. The Committee could have omitted any mention of operators causing conversions, but would then have had to enumerate the behavior for all possible pairs of operand types. The usual arithmetic conversions are the lesser of two evils (reducing the number of different arithmetic types did not get a look in).

The operators referred to here could be either unary or binary operators. In the case of the unary operators the standard specifies no conversion if the operand has a floating type (although the implementation may perform one, as indicated by the value of the `FLT_EVAL_METHOD` macro). However, there is a conversion (the integer promotions) if the operand has an integer type.

354 FLT_EVAL_ME

Other Languages

Numeric promotion is applied to the operands of an arithmetic operator. Java

All languages that support more than one type need to specify the behavior when an operator is given operands whose types are not the same. Invariably the type that can represent the widest range of values tends to be chosen.

Coding Guidelines

Readers needing to comprehend the evaluation of an expression in detail, need to work out which conversions, if any, are carried out as a result of the usual arithmetic conversions (and the integer promotions). This evaluation requires readers to apply a set of rules. Most developers are not exposed on a regular basis to a broad combination of operand type pairs. Without practice, developers' skill in deducing the result of the usual arithmetic conversions on the less commonly encountered combinations of types will fade away (training can only provide a temporary respite). The consequences of this restricted practice is that developers never progress past the stage of remembering a few specific cases.

If a single integer type were used, the need for developers to deduce the result of the usual arithmetic conversions would be removed. This is one of the primary rationales behind the guideline recommendation that only a single integer type be used.

480.1 object int type only
653 operand convert automatically

As with the integer promotions, there is a commonly seen, incorrect assumption made about the usual

arithmetic conversions. This incorrect assumption is that they do not apply to objects defined using typedef names. In fact a typedef name is nothing more than a synonym. How an object is defined makes no difference to how the usual arithmetic conversions are applied. Pointing out this during developer training may help prevent developers from making this assumption.

typedef 1633 is synonym

---

common real type The purpose is to determine a *common real type* for the operands and result. 703

Commentary

This defines the term *common real type*. The basic idea is that both types are converted to the real type capable of holding the largest range of values (mixed signed/unsigned types are the fly in the ointment). In the case of integer types, the type with the largest rank is chosen. For types with the same rank, the conversions are unsigned preserving.

Many processor instructions operate on values that have the same type (or at least are treated as if they did). C recognizes this fact and specifies how the operands are to be converted. This process also removes the need to enumerate the behavior for all possible permutations of operand type pairs.

C90

The term *common real type* is new in C99; the equivalent C90 term was *common type*.

Other Languages

The need to convert the operands to the same type is common to languages that have a large number of different arithmetic types. There are simply too many permutations of different operand type pairs to want to enumerate, or expect developers to remember, the behavior for all cases. Some languages (e.g., PL/1) try very hard to implicitly convert the operands to the same type, without the developer having to specify any explicit conversions. Other languages (e.g., Pascal) enumerate a small number of implicit conversions and require the developer to explicitly specify how any other conversions are to be performed (by making it a constraint violation to mix operands having other type pairs).

Common Implementations

On some processors arithmetic operations can produce a result that is wider than the original operands—for instance multiplying two 16-bit values to give a 32-bit result. In these cases C requires that the result be converted back to the same width as the original operands. Such processor instructions also offer the potential optimization of not performing the widening to 32 bits before carrying out the multiplication. Other operand/result sizes include 8/16 bits.

Coding Guidelines

The C90 term *common type* is sometimes used by developers. Given that few developers will ever use complex types it is unlikely that there will be a general shift in terminology usage to the new C90 term.

Some developers make the incorrect assumption that if the two operands already have a common type, the usual arithmetic conversions are not applied. They forget about the integer promotions. For instance, two operands of type **short** are both promoted to **int** before the usual arithmetic conversions are applied.

Pointing out to developers that the integer promotions are always performed is a training issue (explicitly pointing out this case may help prevent developers from making this assumption). Following the guideline on using a single integer type ensures that incorrect developer assumptions about this promotion do not affect the intended behavior.

object 480.1 int type only

Example

```
1  unsigned char c1, c2, c3;
2
3  int f(void)
4  {
5  /*
6   * Experience shows that many developers expect the following additional and
```

```
7    * relational operations to be performed on character types, rather than int.
8    */
9   if ((c1 + c2) > c3)
10     c1 = 3;
11  }
```

704 For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type.

arithmetic conversions type domain unchanged

Commentary

This requirement means that the usual arithmetic conversions leave operands that have complex types as complex types and operands that have real types remain real types for the purposes of performing the operation. As well as saving the execution-time overhead on the conversion and additional work for the operator, this behavior helps prevent some unexpected results from occurring. The following example first shows the effects of a multiplication using the C99 rules:

500 complex types

520 real types

$$2.0 * (3.0 + \infty i) \Rightarrow 2.0 * 3.0 + 2.0 * \infty i \quad (704.1)$$

$$\Rightarrow 6.0 + \infty i \quad (704.2)$$

The result, $6.0 + \infty i$, is what the developer probably expected. Now assume that the usual arithmetic conversions were defined to change the type domain of the operands, a real type having $0.0i$ added to it when converting to a complex type. In this case, we get:

$$2.0 * (3.0 + \infty i) \Rightarrow (2.0 + 0.0i) * (3.0 - \infty i) \quad (704.3)$$

$$(2.0 * 3.0 - 0.0 * \infty) + (2.0 * \infty + 0.0 * 3.0)i \Rightarrow NaN + \infty i \quad (704.4)$$

The result, $NaN + \infty i$, is probably a surprise to the developer. For imaginary types:

$$2.0i * (\infty + 3.0i) \quad (704.5)$$

leads to $NaN + \infty i$ in one case and $-6.0 + \infty i$ in the C99 case.

C90

Support for type domains is new in C99.

C++

The term *type domain* is new in C99 and is not defined in the C++ Standard.

The template class `complex` contain constructors that can be used to implicitly convert to the matching complex type. The operators defined in these templates all return the appropriate complex type.

C++ converts all operands to a complex type before performing the operation. In the above example the C result is $6.0 + \infty i$, while the C++ result is $NaN + \infty i$.

Other Languages

Fortran converts the operand having the real type to a complex type before performing any operations.

Coding Guidelines

Support for complex types is new in C99, and at the time of this writing there is very little practical experience available on the sort of mistakes that developers make with it. An obvious potential misunderstanding would to be assume that if one operand has a complex type then the other operand will also be converted to the corresponding complex type. This thinking fits the pattern of the other conversions, but would be incorrect. Based on the same rationale as that given in the previous two sentences, the solution is training, not a guideline recommendation.

arithmetic conversions result type

Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. 705

Commentary

The only place where the standard *explicitly stated otherwise* is in the discussion of imaginary types in annex G. Support for such types, by an implementation, is optional. When one operand has a complex type and the other operand does not, the latter operand is not converted to a different type domain (although its real type may be changed by a conversion), so there is no common arithmetic type, only a common real type.

footnote 719 51

C++

The `complex` specializations (26.2.3) define conversions for **`float`**, **`double`** and **`long double`** to complex classes. A number of the constructors are defined as explicit, which means they do not happen implicitly, they can only be used explicitly. The effect is to create a different result type in some cases.

In C++, if the one operand does not have a complex type, it is converted to the corresponding complex type, and the result type is the same as the other operand having complex type. See footnote 51.

footnote 719 51

Other Languages

The result type being the same as the final type of the operands is true in most languages.

usual arithmetic conversions

This pattern is called the *usual arithmetic conversions*: 706

Commentary

This defines the term *usual arithmetic conversions*. There are variations on this term used by developers; however, *arithmetic conversions* is probably the most commonly heard.

Other Languages

The equivalent operations in Java are known as the *Binary Numeric Promotion*.

Common Implementations

In some cases the standard may specify two conversions— an integer promotion followed by an arithmetic conversion. An implementation need not perform two conversions. The as-if rule can be used to perform a single conversion (if the target processor has such an instruction available to it) provided the final result is the same.

Coding Guidelines

The concept of usual arithmetic conversions is very important in any source that has operands of different types occurring together in the same expression. The guideline recommendation specifying use of a single integer type is an attempt to prevent this from occurring. The general issue of whether any operand that is converted should be explicitly cast, rather than implicitly converted, is discussed elsewhere.

object 480.1 int type only
operand 653 convert automatically

arithmetic conversions long double

First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**. 707

Commentary

If the other operand has an integer type, the integer promotions are not first applied to it.

integer promotions 675

Other Languages

The operands having floating types takes precedence over those having integer types in all languages known to your author.

Common Implementations

Processors often support an instruction for converting a value having a particular integer type to a floating representation. Implementations have to effectively promote integer values to this type before they can be converted to a floating type. Since the integer type is usually the widest one available, there is rarely an externally noticeable difference.

Coding Guidelines

If both operands have a floating type, there is no loss of precision on the conversion.

When an integer is converted to a real type, there may be some loss of precision. The specific case of integer-to-floating conversion might be considered different from integer-to-integer and floating-to-floating conversions for a number of reasons:

- There is a significant change of representation.
- The mixing of operands having an integer and floating type is not common (the implication being that more information will be conveyed by an explicit cast in this context than in other contexts).
- The decision to declare an object to have a floating type is a much bigger one than giving it an integer type (experience suggests that, once this decision is made, it is much less likely to change than had an integer type has been chosen, for which there are many choices; so the maintenance issue of keeping explicit conversions up to date might be a smaller one than for integer conversions).

695 float promoted to double or long double
688 integer conversion to floating
688 integer conversion to floating

---

708 Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**.

arithmetic conversions double

Commentary

The operand may already be represented during expression evaluation to a greater precision than the type **double** (perhaps even type **long double**). However, from the point of view of type compatibility and other semantic rules, its type is still **double**.

354 FLT_EVAL_ME

Coding Guidelines

The issues associated with the operands being represented to a greater precision than denoted by their type are discussed elsewhere.

If the other operand has an integer type and is very large there is a possibility that it will not be exactly represented in type **double**, particularly if this type only occupies 32 bits (a single-precision IEC representation, the minimum required by the C Standard). The issues associated with of exact representation are discussed elsewhere.

354 FLT_EVAL_ME
688 integer conversion to floating

---

709 Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.[51)]

arithmetic conversions float

Commentary

As was pointed out in the previous sentence, when one of the operands has type **double**, the operand may be represented to a greater precision during expression evaluation.

708 arithmetic conversions double

Common Implementations

Many implementations perform operations on floating-point operands using the type **double** (either because the instruction that loads the value from storage converts them to this form, or because the translator generates an instruction to implicitly convert them). The Intel x86 processor[627] implicitly converts values to an internal 80-bit form when they are loaded from storage.

---

710 Otherwise, the integer promotions are performed on both operands.

arithmetic conversions integer promotions

Commentary

At this point the usual arithmetic conversions become a two-stage process. Whether an implementation actually performs two conversions, or can merge everything into a single conversion, is an optimization issue that is not of concern to the developer. Performing the integer promotions reduces the number of permutations of operand types that the standard needs to specify behavior for.

675 integer promotions

Other Languages

Most languages have a small number of arithmetic types, and it is practical to fully enumerate the behavior in all cases, making it unnecessary to define an intermediate step.

Coding Guidelines

integer promotions 675
operand same type 712
no further conversion
object int type only 480.1

Developers sometimes forget about this step (i.e., they overlook the fact that the integer promotions are performed). The rule they jump to is often the one specifying that there need be no further conversions if the types are the same. There is no obvious guideline recommendation that can render this oversight (or lack of knowledge on the part of developers) harmless. If the guideline recommendation specifying use of a single integer type is followed these promotions will have no effect.

Usage

Usage information on integer promotions is given elsewhere (see Table 675.1).

---

arithmetic conversions integer types

Then the following rules are applied to the promoted operands: 711

Commentary

The C90 Standard enumerated each case based on type. Now the that concept of rank has been introduced, it is possible to specify rules in a more generalized form. This generalization also handles any extended integer types that an implementation may provide.

C++

The rules in the C++ Standard appear in a bulleted list of types with an implied sequential application order.

Common Implementations

In those implementations where the two types have the same representation no machine code, to perform the conversion at execution time, need be generated.

While the standard may specify a two-stage conversion process, some processors support instructions that enable some of the usual arithmetic conversions to be performed in a single instruction.

Coding Guidelines

standard integer types 493
conversion rank 659

Developers have a lot of experience in dealing with the promotion and conversion of the standard integer types. In the C90 Standard the rules for these conversions were expressed using the names of the types, while C99 uses the newly created concept of rank. Not only is the existing use of extended integer types rare (and so developers are unlikely to be practiced in their use), but conversions involving them use rules based on their rank (which developers will have to deduce). At the time of this writing there is insufficient experience available with extended integer types to be able to estimate the extent to which the use of operands having some extended type will result in developers incorrectly deducing the result type. For this reason these coding guidelines sections say nothing more about the issue (although if the guideline recommendation specifying use of a single integer type is followed the promotion of extended integer types does not become an issue).

object int type only 480.1

For one of the operands, these conversions can cause either its rank to be increased, its sign changed, or both of these things. An increase in rank is unlikely to have a surprising affect (unless performance is an issue, there can also be cascading consequences in that the result may need to be converted to a lesser rank later). A change of sign is more likely to cause a surprising result to be generated (i.e., a dramatic change in the magnitude of the value, or the reversal of its signedness). While 50% of the possible values an object can hold may produce a surprising result when it is converted between signed and unsigned, in practice the values that occur are often small and positive (see Table 940.4).

Experience has shown that mixing operands having signed and unsigned types in expressions has a benefit. Despite a great deal of effort, by a large number of people, no guideline recommendation having a cost less than that incurred by the occasional surprising results caused by the arithmetic conversions has been found.

Usage

Usage information on implicit conversions is given elsewhere (see Table 653.1).

712 If both operands have the same type, then no further conversion is needed.

operand same type no further conversion

Commentary

Both operands can have the same type because they were declared to have that type, the integer promotions converted them to that type, or because they were explicitly cast to those types.

C90

For language lawyers only: A subtle difference in requirements exists between the C90 and C99 Standard (which in practice would have been optimized away by implementations). The rules in the C90 wording were ordered such that when two operands had the same type, except when both were type **int**, a conversion was required. So the type **unsigned long** needed to be converted to an **unsigned long**, or a **long** to a **long**, or an **unsigned int** to an **unsigned int**.

Coding Guidelines

Many developers incorrectly assume this statement is true for all integer types, even those types whose rank is less than that of type **int**. They forget, or were never aware, that the integer promotions are always performed. The guideline recommendation dealing with a single integer type aims to ensure that both operands always have the same type.

675 integer promotions
480.1 object int type only

713 Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

Commentary

If the operands are both signed, or both unsigned, it is guaranteed that their value can be represented in the integer type of greater rank. This rule preserves the operand value and its signedness.

494 rank relative ranges

Other Languages

Those languages that support different size integer types invariably also convert the *smaller* integer type to the *larger* integer type.

Common Implementations

This conversion is usually implemented by sign-extending the more significant bits of the value for a signed operand, and by zero-filling for an unsigned operand. Many processors have instructions that will sign-extend or zero-fill a byte, or half word value, when it is loaded from storage into a register. Some processors have instructions that take operands of different widths; for instance, multiplying a 16-bit value by an 8-bit value. In these cases it may be possible to optimize away the conversion and to use specific instructions that return the expected result.

714 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

signed integer converted to unsigned

Commentary

This rule could be said to be unsigned preserving. The behavior for conversion to an unsigned type is completely specified by the standard, while the conversion to a signed type is not.

683 unsigned integer conversion to
684 integer value not represented in signed integer

Other Languages

Languages that have an unsigned type either follow the same rules as C in this situation, or require an explicit conversion to be specified (e.g., Modula-2).

Coding Guidelines

A signed operand having a negative value will be converted to a large unsigned value, which could be a surprising result. A guideline recommending against operands having negative values, in this case, might be considered equivalent to one recommending against faults— i.e., pointless. The real issue is one of operands of different signed'ness appearing together within an expression. This issue is discussed elsewhere.

683 unsigned integer conversion to
486 signed integer corresponding unsigned integer

Example

width 626
integer type

The width of the integer types is irrelevant. The conversion is based on rank only. The unsigned type has greater rank and the matching rule has to be applied.

```
1  signed int si;
2  unsigned long ul;
3
4  void f(void)
5  {
6  /* ... */ si + ul;
7  }
```

---

signed integer represent all unsigned integer values

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type. 715

Commentary

This rule is worded in terms of representable values, not rank. It preserves the value of the operand, but not its signedness. While a greater range of representable values does map to greater rank, the reverse is not true. It is possible for types that differ in their rank to have the same range of representable values. The case of same representation is covered in the following rule.

Other Languages

Most languages have a single integer type. Those that support an unsigned type usually only have one of them. A rule like this one is very unlikely to be needed.

Common Implementations

In this case an implementation need not generate any machine code. The translator can simply regard the value as having the appropriate signed integer type.

Example

In the following addition two possible combinations of representation yield the conversions:

1. 16-bit unsigned int plus 32-bit long $\Rightarrow$ 32-bit long
2. 32-bit unsigned int plus 32-bit long $\Rightarrow$ this rule not matched

```
1  unsigned int ui;
2  signed long sl;
3
4  void f(void)
5  {
6  /* ... */ ui + sl;
7  }
```

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type. 716

Commentary

This rule is only reached when one operand is an unsigned integer type whose precision is greater than or equal to the precision of a signed type having a greater rank. This can occur when two standard types have the same representation, or when the unsigned type is an extended unsigned integer type (which must have a

lesser rank than any standard integer type having the same precision). In this rule the result type is different from the type of either of the operands. The unsignedness is taken from one operand and the rank from another to create the final result type. This rule is the value-driven equivalent of the rank-driven rule specified previously.

714 signed integer converted to unsigned

Coding Guidelines

This result type of this rule is likely to come as a surprise to many developers, particularly since it is dependent on the width and precision of the operands (because of the previous rule). Depending on the representation of integer types used, the previous rule may match on one implementation and this rule match on another. The result of the usual arithmetic conversions, in some cases, is thus dependent on the representation of integer types.

A previous rule dealt with operands being converted from a signed to an unsigned type. For the rule specified here, there is the added uncertainty of the behavior depending on the integer representations used by an implementation.

714 signed integer converted to unsigned

Example

```
1  #include <stdint.h>
2
3  uint_fast32_t fast_ui;
4  signed long sl;
5  unsigned int ui;
6
7  void f(void)
8  {
9  /*
10  * 16-bit unsigned int + 32-bit signed long -> 32-bit signed long
11  * 32-bit unsigned int + 32-bit signed long -> 32-bit unsigned long
12  */
13 ui + sl;
14
15 fast_ui + sl; /* unsigned long result. */
16 }
```

---

717 The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type;

Commentary

This issue is discussed under the FLT_EVAL_METHOD macro.

354 FLT_EVAL_ME

Other Languages

Most languages do not get involved in specifying this level of detail (although Ada explicitly requires the precision to be as defined by the types of the operands).

Coding Guidelines

Additional precision does not necessarily make for more accurate results, or more consistent behavior (in fact often the reverse). The issues involved are discussed in more detail under the FLT_EVAL_METHOD macro.

354 FLT_EVAL_ME

---

718 the types are not changed thereby.[52)]

Commentary

Any extra precision that might be held by the implementation does not affect the semantic type of the result.

---

719 51) For example, addition of a **double _Complex** and a **float** entails just the conversion of the **float** operand to **double** (and yields a **double _Complex** result).

footnote 51

Commentary

Similarly, the addition of operands having types **float _Complex** and **double** entails conversion of the complex operand to **double _Complex**.

C90

Support for complex types is new in C99

C++

arithmetic conversions 709
float

The conversion sequence is different in C++. In C++ the operand having type **float** will be converted to `complexfloat` prior to the addition operation.

```
1  #include <complex.h> // the equivalent C++ header
2
3  float complex fc; // std::complex<float> fc;  this is the equivalent C++ declaration
4  double d;
5
6  void f(void)
7  {
8  fc + d   /* Result has type double complex. */
9           // Result has type complex<float>.
10       ;
11 }
```

footnote 52

52) The cast and assignment operators are still required to perform their specified conversions as described in 6.3.1.4 and 6.3.1.5. 720

Commentary

To be exact an implementation is required to generate code that behaves as if the specified conversions were performed. An implementation can optimize away any conversion if it knows that the current value representation is exactly the same as the type converted to (implicitly or explicitly). When an implementation carries out floating-point operations to a greater precision than required by the semantic type, the cast operator is guaranteed to return a result that contains only the precision required by the type converted to. An explicit cast provides a filter through which any additional precision cannot pass.

The assignment operator copies a value into an object. The object will have been allocated sufficient storage to be able to hold any value representable in its declared type. An implementation therefore has to scrape off any additional precision which may be being held in the value (i.e., if the register holding it has more precision), prior to storing a value into an object.

Passing an argument in a function call, where a visible prototype specifies a parameter type of less floating-point precision, is also required to perform these conversions.

Other Languages

A universal feature of strongly typed languages is that the assignment operator is only able to store a value into an object that is representable in an object's declared type. Many of these languages do not get involved in representation details. They discuss the cast operator, if one is available, in terms of change of type and tend to say nothing about representation issues.

In some languages the type of the value being assigned becomes the type of the object assigned to. For such language objects are simply labels for values and have no identity of their own, apart from their name (and they may not even have that).

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

lvalue

An *lvalue* is an expression with an object type or an incomplete type other than **void**;[53] 721

Commentary

This defines the term *lvalue*, pronounced *ell-value*. It is an important concept and this term is used by the more sophisticated developers. An lvalue can have its address taken.

The difference between evaluating, for instance, the expression `a[i]` as a value and as an lvalue is that in the former case the result is the value, while in the latter it is the address of the i'th element of the array `a`.

```
From: Dennis Ritchie
Newsgroups: comp.std.c
Subject: Re: C99 change of lvalue definition
Date: Sat, 27 Oct 2001 03:32:09 +0000
Organization: Bell Labs / Lucent Technologies
 . . .
> Part of the problem is that the term "lvalue" is being used for two
> different notions, one syntactic (a certain class of expressions) and
> the other semantic (some abstraction of the notion of "address").  The
> first step would be to remove this ambiguity by selecting distinct
> terms for these two notions, e.g., "lvalue-expression" for the
> syntactic notion and "lvalue" for the semantic notion.
>
> The syntactic notion could be defined by suitably amending the grammar
> appendix so that lvalue-expression is a non-terminal whose expansions
> are exactly the expressions that can appear on the left side of an
> assignment without causing a compile-time error.
>
  . . .
> Tom Payne
This is true.  To clean up the history a bit: The 1967 BCPL manual I
have uses the words "lvalue" and "rvalue" liberally; my copy does not
have a consolidated grammar, but the section about Simple Assignment
Commands describes the syntactical possibilities for things on the
left of := , namely an identifier (not a manifest constant), a vector
application, or an rv expression (equivalent to C *, or later BCPL
!).  However, "lvalue" in the text does indeed seem to be a semantic
notion, namely a bit pattern that refers to a cell in the abstract
machine.
The Richards and Whitbey-Strevens book (my printing seems post-1985,
but it's a reprint of the 1980 edition) does not seem to use the
terms lvalue or rvalue.  On the other hand, it does make even more
explicit syntactically (in its grammar) what can appear on the LHS of
the := of an assignment-command, namely an <lhse> or left-hand-side
expression. This is the syntactic lvalue.
On yet another hand, it only indirectly and by annotation says that
the operand of the address-of operator is restricted.  Nevertheless,
the textual description seems identical to the consolidated grammar's
<lhse>.
In any event, my observation that K&R 1 used syntactic means as the
underlying basis for answering the question "what can go on the left
of = or as the operand of ++, &, ...?" seems true.  In C89 and C99
there are no syntactical restrictions; the many that exist are
semantic, for better or worse.
Dennis
```

C90

> *An lvalue is an expression (with an object type or an incomplete type other than* **void***) that designates an object.31)*

assignment operator 1289
modifiable lvalue

The C90 Standard required that an lvalue designate an object. An implication of this requirement was that some constraint requirements could only be enforced during program execution (e.g., the left operand of an assignment operator must be an lvalue). The Committee intended that constraint requirements be enforceable during translation.

Technically this is a change of behavior between C99 and C90. But since few implementations enforced this requirement during program execution, the difference is unlikely to be noticed.

C++

3.10p2 *An lvalue refers to an object or function.*

object types 475

Incomplete types, other than **void**, are object types in C++, so all C lvalues are also C++ lvalues.

The C++ support for a function lvalue involves the use of some syntax that is not supported in C.

3.10p3 *As another example, the function*

```
int& f();
```

yields an lvalue, so the call `f()` *is an lvalue expression.*

Other Languages

While not being generic to programming languages, the concept of lvalue is very commonly seen in the specification of other languages.

---

if an lvalue does not designate an object when it is evaluated, the behavior is undefined. 722

Commentary

This can occur if a dereferenced pointer does not refer to an object, or perhaps it refers to an object whose lifetime has ended.

lifetime of object 451

C90

In the C90 Standard the definition of the term *lvalue* required that it designate an object. An expression could not be an lvalue unless it designated an object.

C++

In C++ the behavior is not always undefined:

3.8p6 *Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.3.2), and using the properties of the lvalue which do not depend on its value is well-defined.*

Coding Guidelines

A guideline recommending that an lvalue always denote an object when it is evaluated is equivalent to a guideline recommending that programs not contain defects.

Example

```
1  extern int *p;
2
3  void f(void)
4  {
5  p[3]=3; /* Does p designate an object? */
6  }
```

---

723 When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. particular type

Commentary

This is not a definition of the term *particular type*, which is only used in two other places in the standard. In the case of objects with static and automatic storage duration, it is usually the declared type of the object. For objects with allocated storage duration, almost any type can be used. The term *effective type* was introduced in C99 to provide a more precise specification of an object's type. 948 effective type

C++

The situation in C++ is rather more complex:

> *The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do.] An* `object` *is created by a* `definition` *(3.1), by a* `new-expression` *(5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a* `name` *(clause 3). An object has a* `storage duration` *(3.7) which influences its* `lifetime` *(3.8). An object has a* `type` *(3.9). The term* `object type` *refers to the type with which the object is created. Some objects are* `polymorphic` *(10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the* `expressions` *(clause 5) used to access them.* 1.8p1

Example

```
1  extern int ei;
2  extern void *pv;
3
4  void f(void)
5  {
6  ei = 2;
7  *(char *)pv = 'x';
8  }
```

---

724 A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type. modifiable lvalue

Commentary

This defines the term *modifiable lvalue*; which is not commonly used by developers. Since most lvalues are modifiable, the term *lvalue* tends to be commonly used to denote those that are modifiable. There are

a variety of different terms used to describe lvalues that are not modifiable lvalues, usually involving the phrase *const-qualified*. As the name suggests, a modifiable lvalue can be modified during program execution. All of the types listed in the C sentence are types whose lvalues are not intended to be modified, or for which there is insufficient information available to be able to modify them (an incomplete type).

footnote 1131 85 The result of a cast is not an lvalue, so it is not possible to cast away a const-qualification. However, it is possible to cast away constness via a pointer type:

```
1  struct {
2          const int m1;
3          /*
4           * The const qualifier applies to the array
5           * element, not to the array a_mem.
6           */
7          const int a_mem[3];
8          } glob;
9
10 const int * cp = &glob.m1;
11
12 void f(void)
13 {
14 *(int *)cp=2;
15 }
```

The term member applies to structure and union types, while element refers to array types.

C++

The term *modifiable lvalue* is used by the C++ Standard, but understanding what this term might mean requires joining together the definitions of the terms *lvalue* and *modifiable*:

3.10p10 *An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances.*

3.10p14 *If an expression can be used to modify the object to which it refers, the expression is called modifiable.*

There does not appear to be any mechanism for modifying objects having an incomplete type.

8.3.4p5 *Objects of array types cannot be modified, see 3.10.*

This is a case where an object of a given type cannot be modified and follows the C requirement.

7.1.5.1p3 *. . . ; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.*

The C++ wording is based on access paths rather than the C method of enumerating the various cases. However, the final effect is the same.

Other Languages

Many languages only provide a single way of modifying an object through assignment. In these cases a general term describing modifiability is not required; the requirements can all be specified under assignment. Languages that support operators that can modify the value of an object, other than by assignment, sometimes define a term that serves a purpose similar to modifiable lvalue.

---

lvalue converted to value

Except when it is the operand of the **sizeof** operator, the unary **&** operator, the **++** operator, the **--** operator, or the left operand of the **.** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). 725

Commentary

The exceptions called out here apply either because information on the storage for an object is used or are situations where there is a possibility for the lvalue to be modified (in the case of the **&** and **.** operators this may occur in a subsequent operation). In the case of an array type, in most contexts, a reference to an object having this type is converted to a pointer to its first element (and so is always an lvalue). This converted lvalue is sometimes called an *rvalue*.

729 array converted to pointer

736 rvalue

C++

Quite a long chain of deduction is needed to show that this requirement also applies in C++. The C++ Standard uses the term *rvalue* to refer to the particular value that an lvalue is converted into.

> *Whenever an lvalue appears in a context where an rvalue is expected, the lvalue is converted to an rvalue;* 3.10p7

> *Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, the lvalue-to-rvalue (4.1), . . . standard conversions are applied to convert the expression to an rvalue.* 5p8

The C wording specifies that lvalues are converted unless they occur in specified contexts. The C++ wording specifies that lvalues are converted in a context where an rvalue is expected. Enumerating the cases where C++ expects an rvalue we find:

> *The lvalue-to-rvalue (4.1), . . . standard conversions are not applied to the operand of* ***sizeof****.* 5.3.4p4

What is the behavior for the unary **&** operator?

> *There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary* ***&*** *operator.* 4p5

However, this is a *Note:* and has no normative status. There is no mention of any conversions in 5.3.1p2-5, which deals with the unary **&** operator.

In the case of the postfix **++** and **--** operators we have:

> *The operand shall be a modifiable lvalue. . . . The result is an rvalue.* 5.2.6p1

In the case of the prefix **++** and **--** operators we have:

> *The operand shall be a modifiable lvalue. . . . The value is the new value of the operand; it is an lvalue.* 5.3.2p1

So for the postfix case, there is an lvalue-to-rvalue conversion, although this is never explicitly stated and in the prefix case there is no conversion.

The C case is more restrictive than C++, which requires a conforming implementation to successfully translate:

```
1  extern int i;
2
3  void f(void)
4  {
5  ++i = 4; // Well-formed
6           /* Constraint violation */
7  }
```

For the left operand of the **.** operator we have:

5.2.5p4

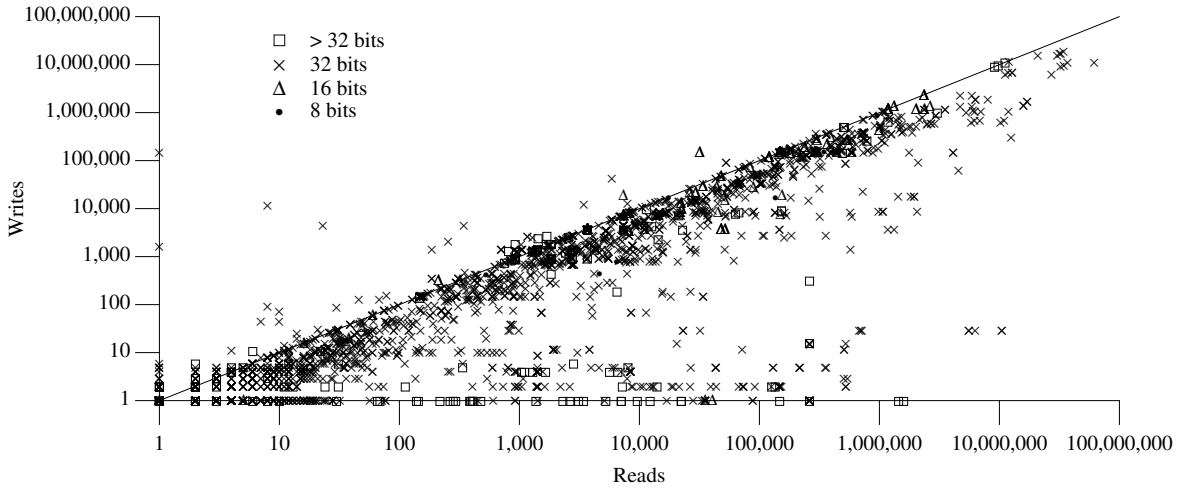


Figure 725.1: Execution-time counts of the number of reads and writes of the same object (declared in block or file scope, i.e., not allocated storage) for a subset of the MediaBench benchmarks; items above the diagonal indicate more writes than reads. Data kindly supplied by Caspi, based on his research.[207]

If `E1` *is an lvalue, then* `E1.E2` *is an lvalue.*

The left operand is not converted to an rvalue. For the left operand of an assignment operator we have:

5.17p1 *All require a modifiable lvalue as their left operand, . . . ; the result is an lvalue.*

The left operand is not converted to an rvalue. And finally for the array type:

4.1p1 *An lvalue (3.10) of a non-function, non-array type* `T` *can be converted to an rvalue.*

An lvalue having an array type cannot be converted to an rvalue (i.e., the C++ Standard contains no other wording specifying that an array can be converted to an rvalue).

In two cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clause 5.18p1 left operand of the comma operator and Clause 6.2p1 the expression in an expression statement. In C the values would be discarded in both of these cases, so there is no change in behavior. In the following cases C++ performs a lvalue-to-rvalue conversion (however, the language construct is not relevant to C): Clause 8.2.8p3 Type identification; 5.2.9p4 static cast; 8.5.3p5 References.

Other Languages

The value that the lvalue is converted into is sometimes called an *rvalue* in other languages.

Coding Guidelines

The distinction between lvalue and rvalue and the circumstances in which the former is converted into the latter is something that people learning to write software for the first time often have problems with. But once the underlying concepts are understood, developers know how to distinguish the two contexts. The confusion that developers often get themselves into with pointers is the *pointer* or the *pointed-to object* being accessed is a separate issue and is discussed under the indirection operator.

unary * 1095
indirection

Usage

Usage information on the number of translation time references, in the source code, is given elsewhere (see Figure 1821.5, Figure 1821.6).

---

lvalue
value is unqualified

If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; 726

Commentary

Type qualification only applies to objects, not to values. A pointer value can refer to an object having a qualified type. In this case it is not the pointer value that is qualified.

```
1  extern int glob;
2  extern const int *p_1;
3  extern       int * const p_2 = &glob; /* The value of p_2 cannot be modified. */
4
5  void f(void)
6  {
7  const int loc = 2;
8
9  p_1 = &loc; /* The value of p_1 can be modified. */
10 }
```

C++

The value being referred to in C is what C++ calls an *rvalue*.

If T is a non-class type, the type of the rvalue is the cv-unqualified version of T. Otherwise, the type of the rvalue is T.[49)] 4.1p1

49) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types. Footnote 49

Class rvalues are objects having a structure or union type in C.

Other Languages

Some languages give string literals their equivalent of a **const** qualifier, thus making it possible for values to be qualified, as well as have a type.

---

727 otherwise, the value has the type of the lvalue.

Commentary

Accessing an lvalues value does not change its type, even if there are more bits in the object representation than in the value representation. Those values that have an integer type whose rank is less than that of **int** may be promoted to **int** or some other type as a result of being the operand of some operator. But values start off with the type of the lvalue from which they were obtained.

574 object representation
595 value representation
675 integer promotions

Other Languages

This statement is probably generic to all languages.

---

728 If the lvalue has an incomplete type and does not have array type, the behavior is undefined.

Commentary

In the case of array types the element type is known, which is the only information needed by a translator (to calculate the offset of the indexed element, given any index value).

As the standard points out elsewhere, an incomplete type may only be used when the size of an object of that type is not needed.

1465 footnote 109

C++

4.1p1

An lvalue (3.10) of a non-function, non-array type T can be converted to an rvalue. If T is an incomplete type, a program that necessitates this conversion is ill-formed.

4.2p1 *An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to an rvalue of type "pointer to T."*

The C behavior in this case is undefined; in C++ the conversion is ill-formed and a diagnostic is required.

Common Implementations

It is very likely that an occurrence of this undefined behavior will be diagnosed by an implementation.

Coding Guidelines

It may surprise developers to find this case is not a constraint violation, but undefined behavior. While the standard specifies that the behavior is undefined, in this case, it is difficult to see how a translator could do anything other than issue a diagnostic and fail to translate (there are no meaningful semantics to give to such a construct). This is not to say that a low-quality translator will accept a program containing such a construct without issuing a diagnostic. Given the likely translator behavior, no guideline is recommended here.

Example

incomplete array indexing

A reference to an array degenerates into a pointer to its first element:

```
1  int ar[];
2
3  void f(void)
4  {
5  ar[2] = 3;
6  }
```

even though `ar` is incomplete at the time it is indexed. Below is a pointer to an incomplete type:

```
1  struct T *pt, *qt;
2
3  void g(void)
4  {
5  *pt=*qt;
6  }
```

Information on the members of the structure T is not needed when dealing with pointers to that structure.

---

array converted to pointer

Except when it is the operand of the **`sizeof`** operator or the unary **`&`** operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. 729

Commentary

If arrays were converted to a pointer to their first element when appearing as the operand of the **`sizeof`** operator, it would be impossible to obtain their sizes.

array type 1598 adjust to pointer to

A similar conversion occurs in the declaration of parameters having an array type. Almost at the drop of a hat objects having an array type are converted into a pointer to their first element. Thus, complete arrays, unlike structures, cannot be assigned or passed as parameters because an occurrence of the identifier, denoting the array object, is converted to a pointer to the first element of that object.

wide string 905 literal type of

String literals have type array of **`char`**, or array of `wchar_t`.

In many contexts applying the unary **`&`** operator to an operand that is an array index is redundant (rather like using the unary **`+`** operator). However, support for such usage can simplify the automatic generation of C source (by allowing objects having an array type to be treated like any other object type).

```
int a[10];

&a;    /* Type pointer to array of int */
&a[3]; /* Type pointer to int */
a[3];  /* Type int */
```

This sentence in the C99 Standard has relaxed some of the restrictions that were in the C90 Standard:

```
#include <stdio.h>

struct S {
         int a[2];
         };

struct S WG14_N813(void)
{
struct S x;

x.a[0] = x.a[1] = 1;
return x;
}

int main(void)
{
/*
 * The following would have been a constraint violation in C90
 * since the expression WG14_N813(a).a is not an lvalue (in C90 the
 * member selection operator is only an lvalue if its left operand
 * is an lvalue, and a function call is not an lvalue).
 */
printf("Answer is %d\n", WG14_N813().a[0]);
}
```

C90

Except when it is the operand of the **`sizeof`** *operator or the unary* **`&`** *operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with* `wchar_t`*, an lvalue that has type "array of type" is converted to an expression that has type "pointer to type" that points to the initial element of the array object and is not an lvalue.*

The C90 Standard says " . . . , an lvalue that has type "array of type" is converted . . . ", while the C99 Standard says " . . . , an expression that has type . . . ". It is possible to create a non-lvalue array. In these cases the behavior has changed. In C99 the expression `(g?x:y).m1[1]` is no longer a constraint violation (C90 footnote 50, "A conditional expression does not yield an lvalue").

In the following, C90 requires that the size of the pointer type be output, while C99 requires that the size of the array be output.

```
#include <stdio.h>

struct {
       int m1[2];
       } x, y;
int g;

int main(void)
{
printf("size=%ld\n", sizeof((g?x:y).m1));
}
```

C++

5.3.3p4 *The . . . , array-to-pointer (4.2), . . . standard conversions are not applied to the operand of* ***sizeof****.*

4.2p1 *An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to an rvalue of type "pointer to T." The result is a pointer to the first element of the array.*

4.2p2 *A string literal . . . can be converted . . . This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue.*

When is there an explicit appropriate pointer target type? Clause 5.2.1 Subscripting, requires that one of the operands have type *pointer to T*. A character string literal would thus be converted in this context.

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

8.3.5p3 *After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively.*

Clause 5.3.1p2 does not say anything about the conversion of the operand of the unary **&** operator. Given that this operator requires its operand to be an lvalue not converting an lvalue array to an rvalue in this context would be the expected behavior.

There may be other conversions, or lack of, that are specific to C++ constructs that are not supported in C.

Other Languages

Implicitly converting an array to a pointer to its first element is unique to C (and C++).

Common Implementations

Most implementations still follow the C90 behavior. `gcc`'s support for `(g?x:y).m1[1]` is a known extension to C90 that is not an extension in C99.

Coding Guidelines

The implicit conversion of array objects to a pointer to their first element is a great inconvenience in trying to formulate stronger type checking for arrays in C.

Inexperienced, in the C language, developers sometimes equate arrays and pointers much more closely than permitted by this requirement (which applies to uses in expressions, not declarations). For instance, in:

file_1.c
```
1 extern int *a;
```

file_2.c
```
1 extern int a[10];
```

the two declarations of `a` are sometimes incorrectly assumed by developers to be compatible. It is difficult to see what guideline recommendation would overcome incorrect developer assumptions (or poor training). If the guideline recommendation specifying a single point of declaration is followed, this problem will not occur.

identifier declared in one file 422.1

function designator converted to type 732

Unlike the function designator usage, developers are familiar with the fact that objects having an array type are implicitly converted to a pointer to their first element. Whether applying a unary **&** operator to an operand having an array type provides readers with a helpful visual cue or causes them to wonder about the intent of the author ("what is that redundant operator doing there?") is not known.

Example

```
1   static double a[5];
2
3   void f(double b[5])
4   {
5   double (*p)[5] = &a;
6   double **q = &b;    /* This looks suspicious, */
7
8   p = &b;             /* and so does this.      */
9   q = &a;
10  }
```

730 If the array object has register storage class, the behavior is undefined.

array object register storage class

Commentary

The conversion specified in the previous sentence is implicitly returning the address of the array object. This statement is making it clear what the behavior is in this case (no diagnostic is required). It is a constraint violation to explicitly take the address, using the **&** operator, of an object declared with the **register** storage class (even although in this case there is no explicit address-of operator).

1088 unary & operand constraints

1074 footnote 83

C90

This behavior was not explicitly specified in the C90 Standard.

C++

*A **register** specifier has the same semantics as an **auto** specifier together with a hint to the implementation that the object so declared will be heavily used.*

7.1.1p3

Source developed using a C++ translator may contain declarations of array objects that include the **register** storage class. The behavior of programs containing such declarations will be undefined if processed by a C translator.

Coding Guidelines

Given that all but one uses of an array object defined with register storage class, result in undefined behavior (its appearance as the operand of the **sizeof** operator is defined), there appears to be no reason for specifying this storage class in a declaration. However, the usage is very rare and no guideline recommendation is made.

731 A *function designator* is an expression that has function type.

function designator

Commentary

This defines the term *function designator*, a term that is not commonly used in developer discussions. The phrase *designates a function* is sometimes used.

C++

This terminology is not used in the C++ Standard.

Other Languages

Many languages do not treat functions as types. When they occur in an expression, it is because the name of the function is needed in a function call. A few languages do support function types and allow an expression to evaluate to a function type. These languages usually define some technical terminology to describe the occurrence.

function designator converted to type

Except when it is the operand of the **sizeof** operator[54] or the unary **&** operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*". 732

Commentary

Using the unary **&** operator with function types is redundant (as it also is for array types), but supporting such usage makes the automatic generation of C source code simpler (by allowing objects having pointer-to function type to be treated like any other object type). In:

```
1  extern void f(void);
2
3  void g(void)
4  {
5  void (*pf)(void) = f;
6
7  f();
8  pf();
9  }
```

the last two occurrences of `f` are converted to a pointer-to the function denoted by the declaration at file scope. In the first case this pointer is assigned to the object `pf`. In the second case the pointer value is operated on by the **()** operator, causing the pointed-to function to be called. The definition of `g` could, equivalently (and more obscurely), have been:

```
1  void g(void)
2  {
3  void (*pf)(void) = &f;
4
5  &f();
6  (*pf)();
7  }
```

C++

5.3.3p4 *The . . . , and function-to-pointer (4.3) standard conversions are not applied to the operand of* ***sizeof****.*

5.3.1p2 *The result of the unary* ***&*** *operator is a pointer to its operand. The operand shall be an lvalue or a* `qualified-id`*. In the first case, if the type of the expression is "T," the type of the result is "pointer to T."*

While this clause does not say anything about the conversion of the operand of the unary **&** operator, given that this operator returns a result whose type is "pointer to T", not converting it prior to the operator being applied would be the expected behavior. What are the function-to-pointer standard conversions?

4.3p1 *An lvalue of function type T can be converted to an rvalue of type "pointer to T."*

5p8 *Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, . . . , or function-to-pointer (4.3) standard conversions are applied to convert the expression to an rvalue.*

In what contexts does an operator expect an rvalue that will cause a function-to-pointer standard conversion?

5.2.2p1

For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.3) is suppressed on the postfix expression), or it shall have pointer to function type.

The suppression of the function-to-pointer conversion is a difference in specification from C, but the final behavior is the same.

If either the second or the third operand has type (possibly cv-qualified) **`void`***, then the . . . , and function-to-pointer (4.3) standard conversions are performed on the second and third operands,* 5.16p2

If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand. 5.17p3

After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively. 8.3.5p3

This appears to cover all cases.

Other Languages

Those languages that support function types provide a variety of different mechanisms for indicating that the address of a function is being taken. Many of them do not contain an address-of operator, but do have an implicit conversion based on the types of the operands in the assignment.

Common Implementations

Most implementations generate code to call the function directly (an operation almost universally supported by host processors), not to load its address and indirectly call it.

Coding Guidelines

The fact that a function designator is implicitly converted to a pointer-to function type is not well-known to developers. From the practical point of view, this level of detail is unlikely to be of interest to them. Using the unary **&** operator allows developers to make their intentions explicit. The address is being taken and the parentheses have not been omitted by accident. Use of this operator may be redundant from the translators point of view, but from the readers point of view, it can be a useful cue to intent. In:

```
1  int f(void)
2  { /* ... */ }
3
4  void g(void)
5  {
6  if (f)  /* Very suspicious. */
7     ;
8  if (f())
9     ;
10 if (&f) /* Odd, but the intent is clear. */
11    ;
12 }
```

Cg 732.1 When the address of a function is assigned or compared against another value, the address-of operator shall be applied to the function designator.

Example

In the following, all the function calls are equivalent:

```
1   void f(void);
2
3   void g(void)
4   {
5   f();
6   (f)();
7   (*f)();
8   (**f)();
9   (**********f)();
10  (&f)();
11  }
```

they all cause the function `f` to be called.

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions **<stddef.h>** (7.17), initialization (6.7.8), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** operator (6.5.3.4), structure and union members (6.5.2.3). 733

---

footnote 53

53) The name "lvalue" comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** is required to be a (modifiable) lvalue. 734

Commentary

Or *left value* of assignment. The *right value* being called the rvalue. This is translator writers' terminology that the C Standard committee has adopted.

C++

The C++ Standard does not provide a rationale for the origin of the term *lvalue*.

Other Languages

Many languages have terms denoting the concept of lvalue and rvalue. Some use these exact terms.

It is perhaps better considered as representing an object "locator value". 735

Commentary

Many languages only allow an object to be modified through assignment. C is much more flexible, allowing objects to be modified by operators other than assignment (in the case of the prefix increment/decrement operators the object being modified appears on the right). Happily, the term *locator* starts with the letter *l* and can be interpreted to have a meaning close to that required.

Coding Guidelines

This term is not in common usage by any C related groups or bodies. The term *lvalue* is best used. It is defined by the C Standard and is also used in other languages.

---

rvalue

What is sometimes called "rvalue" is in this International Standard described as the "value of an expression". 736

Commentary

The term *rvalue* only appears in this footnote. It is not generally used by C developers, while the term *value of expression* is often heard.

C++

The C++ Standard uses the term *rvalue* extensively, but the origin of the term is never explained.

3.10p1

Every expression is either an lvalue or an rvalue.

Other Languages

The term *rvalue* is used in other languages.

Coding Guidelines

The term *value of an expression* is generally used by developers. While the term *rvalue* is defined in the C++ Standard, its usage by developers in that language does not appear to be any greater than in C. There does not seem to be any benefit in trying to change this commonly used terminology.

---

737 An obvious example of an lvalue is an identifier of an object.

Commentary

A nonobvious lvalue is a string literal. An obvious example of an rvalue is an integer constant.

903 string literal static storage duration

---

738 As a further example, if **E** is a unary expression that is a pointer to an object, ***E** is an lvalue that designates the object to which **E** points.

Commentary

And it may, or may not also be a modifiable lvalue.

---

739 54) Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraint in 6.5.3.4.

footnote 54

Commentary

The committee could have specified a behavior that did not cause this constraint to be violated. However, the size of a function definition is open to several interpretations. Is it the amount of space occupied in the function image, or the number of bytes of machine code generated from the statements contained within the function definition? What about housekeeping information that might need to be kept on the stack during program execution? Whichever definition is chosen, it would require a translator to locate the translated source file containing the function definition. Such an operation is something that not only does not fit into C's separate compilation model, but could be impossible to implement (e.g., if the function definition had not yet been written). The Committee could have specified that the size need not be known until translation phase 8 (i.e., link-time) but did not see sufficient utility in supporting this functionality.

1118 sizeof constraints

139 translation phase 8

C++

The C++ Standard does not specify that use of such a usage renders a program ill-formed:

*The **sizeof** operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field.* 5.3.3p1

6.3.2.2 void

---

740 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression.

void expression

Commentary

This defines the term *void expression*. It makes no sense to take an expression having a type that cannot represent any values and cast it to a type that can represent a range of values. As specified here, these requirements do not require a diagnostic to be issued. However, constraint requirements in the definition of the cast operator do require a translator to issue a diagnostic, if a violation occurs.

1134 cast scalar or void type

C++

3.9.1p9 *An expression of type* **void** *shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of* **?:** *(5.16), as the operand of* **typeid**, *or as the expression in a return statement (6.6.3) for a function with the return type* **void**.

The C++ Standard explicitly enumerates the contexts in which a void expression can appear. The effect is to disallow the value of a void expression being used, or explicitly converted, as per the C wording.
The C++ Standard explicitly permits the use of a void expression in a context that is not supported in C:

```
1  extern void g(void);
2
3  void f(void)
4  {
5  return g(); /* Constraint violation. */
6  }
```

5.2.9p4 *Any expression can be explicitly converted to type "cv* **void***."*

Thus, C++ supports the **void** casts allowed in C.

Other Languages

In many languages the assignment token is not an operator; it is a token that appears in an assignment statement. Casting away the result of an assignment is not possible because it does not have one. Also there is often a requirement that functions returning a value (those that don't return a value are often called procedures or subroutines) must have that value used.

Coding Guidelines

It is possible for a program to accidentally make use of a *void value*; for instance, if a function is defined with a **void** return type, but at the point of call in a different translation unit, it is declared to return a scalar type. The behavior in this case is undefined because of the mismatch between definition and declaration at the point of call. If the guideline recommendations on having a single point of declaration is followed this situation will not occur.

function call not compatible with definition 1024
identifier declared in one file 422.1

If an expression of any other type is evaluated as a void expression, its value or designator is discarded. 741

Commentary

A void expression can occur in an expression statement and as the left operand of the comma operator. According to the definition of a void expression, it has type **void**. In the example below the expression statement `x = y` has type **int**, not **void**. But in this context its value is still discarded. In the following:

void expression 740

```
1  int x, y;
2
3  void f(void)
4  {
5  x = y;
6  (void)(x = y);
7
8  x = (y++, y+1);
9  x = ((void)y++, y+1);
10 }
```

all of the statements are expressions, whose value is discarded after the assignment to `x`.

C90

If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded.

The wording in the C90 Standard begs the question, "When is a void expression required"?

C++

There are a number of contexts in which an expression is evaluated as a void expression in C. In two of these cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clauses 5.18p1 left operand of the comma operator, and 6.2p1 the expression in an expression statement. The other context is an explicit cast:

Any expression can be explicitly converted to type "`cv` ***`void`****." The expression value is discarded.* 5.2.9p4

So in C++ there is no value to discard in these contexts. No other standards wording is required.

Other Languages

Most languages require that the value of an expression be used in some way, but then such languages do not usually regard assignment as an operator (which returns a value) or have the equivalent of the comma operator. In many cases the syntax does not support the use of an expression in a context where its value would be discarded.

Common Implementations

For processors that use registers to hold intermediate results during expression evaluation, it is a simple matter of ignoring register contents when there is no further use for its contents. For stack-based architectures the value being discarded has to be explicitly popped from the evaluation stack at some point. (As an optimization, some implementations only reset the evaluation stack when a backward jump or function **`return`** statement is encountered.)

Coding Guidelines

1732 expression statement evaluated as void expression
1314 comma operator left operand

All expression statements return a value that is not subsequently used, as does the left operand of the comma operator. However, all of the information needed for readers to evaluate the usage is visible at the point the statement or operator appears in the source.

In the case of a function returning a value that is not used, the intent may not be immediately visible to readers. Perhaps the function had not originally returned a value and a later modification changed this behavior. One way of indicating, to readers, that the return value is being intentionally ignored, is to use an explicit cast (the exception for library functions is given because the specification of these functions' return type rarely changes).

Cg 741.1 If a function, that does not belong to a standard library, returning a non-void type is evaluated as a void expression its result shall be explicitly cast to **`void`**.

Example

```
1  extern int f(void);
2
3  void g(void)
4  {
5  (void)f();
6  }
```

(A void expression is evaluated for its side effects.) 742

Commentary

redundant code 190 A void expression is not required to have any side effects. The issue of redundant code is discussed elsewhere.

C++

This observation is not made in the C++ Standard.

Example

In the following:

```
1  int i,
2      j;
3
4  extern int f(void);
5
6  void g(void)
7  {
8  i+1;     /* expression 1 */
9  j=4;     /* expression 2 */
10 4+(i=3); /* expression 3 */
11 (void)f();
12 }
```

execution of *expression 1* will not change the state of the abstract machine. It contains no side effects. Execution of *expression 2* will change the state of the abstract machine, but it may not have any effect on subsequent calculations; or, if it does, those calculations may not have any effect on the output of the program, so is a redundant side effect. *expression 3* contains a side effect, but in a nested subexpression. The root operator in *expression 3* does not contain a side effect.

6.3.2.3 Pointers

pointer to void converted to/from

A pointer to **void** may be converted to or from a pointer to any incomplete or object type. 743

Commentary

generic pointer 523 Pointer to **void** is intended to implement the concept of a generic pointer (other terms include abstract pointer, anonymous pointer), which may point at an object having any type. Constraints on pointer conversions are discussed elsewhere.

pointer conversion constraints 1135

C++

5.2.9p10 *An rvalue of type "pointer to* `cv` ***void****" can be explicitly converted to a pointer to object type.*

object types 475 In C++ incomplete types, other than `cv` **void**, are included in the set of object types.

In C++ the conversion has to be explicit, while in C it can be implicit. C source code that relies on an implicit conversion being performed by a translator will not be accepted by a C++ translator.

The suggested resolution to SC22/WG21 DR #137 proposes changing the above sentence, from 5.2.9p10, to:

Proposed change to C++ *An rvalue of type "pointer to* `cv1` ***void****" can be converted to an rvalue of type "pointer to* `cv2` *T", where T is an object type and* `cv2` *is the same cv-qualification as, or greater cv-qualification than,* `cv1`*.*

If this proposal is adopted, a pointer-to qualified type will no longer, in C++, be implicitly converted unless the destination type is at least as well qualified.

Common Implementations

A pointer to **void** is required to have the same representation as a pointer to **char**. On existing processors where there is a difference between pointer representations, it is between pointer-to character types and pointers-to other types. So the above requirement is probably made possible by the previous representation requirement.

558 pointer to void same representation and alignment as

Coding Guidelines

The pointed-to type, of a pointer type, is useful information. It can be used by a translator to perform type checking and it tells readers of the source something about the pointed-to object. Converting a pointer type to pointer to **void** throws away this information, while converting a pointer to **void** to another pointer type adds information. Should this conversion process be signaled by an explicit cast, or is an implicit conversion acceptable? This issue is discussed elsewhere. The main difference between conversions involving pointer to **void** and other conversions is that they are primarily about a change of information content, not a change of value.

653 operand convert automatically

Example

```
1  extern int ei;
2  extern unsigned char euc;
3
4  void f(void *p)
5  {
6  (*(char *)p)++;
7  }
8
9  void q(void)
10 {
11 unsigned char *p_uc = &euc;
12 int *p_i = &ei;
13
14 f(p_uc);
15 f((void *)p_uc);
16
17 f(p_i);
18 f((void *)p_i);
19 }
```

---

744 A pointer to any incomplete or object type may be converted to a pointer to **void** and back again;

pointer converted to pointer to void

Commentary

This is actually a requirement on the implementation that is worded as a permission for the language user. It implies that any pointer can be converted to pointer to **void** without loss of information about which storage location is being referred to. There is no requirement that the two pointer representations be the same (except for character types), only that the converted original can be converted back to a value that compares equal to the original (see next C sentence). This sentence is a special case of wording given elsewhere, except that here there are no alignment restrictions.

558 pointer to void same representation and alignment as
758 pointer converted to pointer to different object or type

The purpose of this requirement is to support the passing of different types of pointers as function arguments, allowing a single function to handle multiple types rather than requiring multiple functions each handling a single type. This single function is passed sufficient information to enable it to interpret the pointed-to object in a meaningful way, or at least pass the pointer on to another function that does. Using a union type would be less flexible, since a new member would have to be added to the union type every time a new pointer type was passed to that function (which would be impossible to do in the interface to a third-party library).

C++

5.2.10p7 *Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.*

The C++ wording is more general than that for C. A pointer can be converted to any pointer type and back again, delivering the original value, provided the relative alignments are no stricter.

Source developed using a C++ translator may make use of pointer conversion sequences that are not required to be supported by a C translator.

Coding Guidelines

Developers who want to convert a pointer-to X into a pointer-to Y usually do so directly, using a single cast. A conversion path that goes via pointer to **void** is not seen as providing a more reliable result (it doesn't) or a more *type correct* way of doing things (it isn't).

A pointer converted to pointer to **void** and subsequently converted to a different pointer type is either unintended behavior by the developer or a case of deliberate type punning by the developer. Guideline recommendations are not intended to deal with constructs that are obviously faults, and the issue of type punning is discussed elsewhere.

guidelines 0 not faults
type punning 531 union

---

converted via pointer to void compare equal

the result shall compare equal to the original pointer. 745

Commentary

This is a requirement on the implementation. It is not the same as one specifying that the representation shall be bit-for-bit identical. The standard says nothing about the internal representation of pointers. It is possible for two pointers to compare equal and have different bit representations.

pointer 590 segmented architecture

C++

5.2.9p10 *A value of type pointer to object converted to "pointer to cv **void**" and back to the original pointer type will have its original value.*

In C++ incomplete types, other than *cv* **void**, are included in the set of object types.

Common Implementations

Implementations don't usually change the representation of a pointer value when it is cast to another pointer type. However, when the target processor has a segmented architecture, there may be a library call that checks that the value is in canonical form. On most implementations, it is very likely that when converted back the result will be bit-for-bit identical to the original value.

pointer 590 segmented architecture

---

pointer converting qualified/unqualified

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; 746

Commentary

This specification is asymmetric. The specification for when a pointer to a q-qualified type is converted to a pointer to a non-q-qualified version of the type is handled by the general wording on converting object and incomplete types. Converting a pointer to a non-q-qualified type to a pointer to a q-qualified version of the type is going to possibly:

pointer 758 converted to pointer to different object or type

- Limit the operations that will be performed on it (for the **const** qualifier).

- Prevent the translator from optimizing away accesses (for the **volatile** qualifier) and reusing values already held in a register.
- Improve the quality of optimizations performed by a translator (for the **restrict** qualifier) or cause undefined behavior.

1513 restrict undefined behavior

When converting from a pointer-to non-q-qualified to a pointer-to q-qualified type, under what circumstances is it possible to guarantee that the following requirements (sometimes known as *type safe* requirements) are met:

1. A program will not attempt to change the value of an object defined using the **const** qualifier.
2. A program will not access an object defined with the **volatile** in a manner that is not a volatile-qualified access.

For instance, allowing the implicit conversion `char **` $\Rightarrow$ `char const **` would violate one of the above requirements. It could lead to a const-qualified object being modified:

```
1  char const    c  = 'X';
2  char        *pc       ;
3  char const **pcc = &pc; /* Assume this was permitted. */
4  *pcc = &c;
5  *pc = 'Y';              /* Modifies c, a const-qualified object. */
```

This example shows that simply adding the **const** qualifier within a pointer type does not automatically ensure type safety.

Smith[1260] formally proved the necessary conditions that need to hold for the two requirements listed above to be met (provided the type system was not overridden using an explicit cast). Let `T1` and `T2` be the following types, respectively:

$$T cv_{1,n} * \ldots cv_{1,1} * cv_{1,0} \tag{746.1}$$

$$T cv_{2,n} * \ldots cv_{2,1} * cv_{2,0} \tag{746.2}$$

where cv is one of nothing, **const**, **volatile**, or **const volatile**. If $T1$ and $T2$ are different, but similar pointer types, both requirements listed above are met if the following conditions are also met:

1. For every $j > 0$, if **const** is in $cv_{1,j}$, then **const** is in $cv_{2,j}$; and if **volatile** is in $cv_{1,j}$, then **volatile** is in $cv_{2,j}$.
2. If $cv_{1,j}$ and $cv_{2,j}$ are different, then **const** is in every $cv_{2,k}$ for $0 < k < j$.

The first condition requires the converted-to type to contain at least the same qualifiers as the type being converted from for all but the outermost pointer type. For the second case, $cv_{1,j}$ and $cv_{2,j}$ are different when the converted-to type contains any qualifier not contained in $cv_{2,j}$. The requirement on $cv_{2,j}$ including the **const** qualifier ensures that, once converted, a volatile-qualified value cannot be accessed.

```
1  char **    ⇒   char const * const *                  /* Is type safe.     */
2  char **    ⇒   char volatile * volatile *            /* Is not type safe. */
3  char **    ⇒   char const volatile * const volatile *  /* Is type safe.     */
```

There is an important difference of intent between the two qualifiers **const** and **volatile**, and the qualifier **restrict**. The **restrict** qualifier is not subject to some of the constraints on type qualifiers in conversions and assignments of pointers that apply to the other two qualifiers. This is because performing the semantically appropriate checks on uses of the **restrict** qualifier is likely to cause a significant overhead for

restrict 1490 requires all accesses

implementations. The burden of enforcement is placed on developers, to use **restrict** only when they know that the conditions it asserts hold true.

Applying the ideas of type safety, when assigning pointers, to restrict-qualified types leads to the conclusion that it is *safe* to assign to a less **restrict** qualified type (in the formalism used above $j = 0$ rather than $j > 0$) and *unsafe* to assign to a more restrict-qualified type (the opposite usage to that applying to uses of const- or volatile-qualified types). However, it is the *unsafe* conversions that are likely to be of most use in practice (e.g., when passing arguments to functions containing computationally intensive loops).

C++

4.4p1 *An rvalue of type "pointer to* `cv1 T` *" can be converted to an rvalue of type "pointer to* `cv2 T` *" if "* `cv2 T` *" is more cv-qualified than "* `cv1 T` *."*

4.4p2 *An rvalue of type "pointer to member of X of type* `cv1 T` *" can be converted to an rvalue of type "pointer to member of X of type* `cv2 T` *" if "* `cv2 T` *" is more cv-qualified than "* `cv1 T` *."*

Other Languages

The qualifiers available in C do not endow additional access permissions to objects having that type, they only reduce them. This situation is not always true in other languages where specific cases sometimes need to be called out.

Coding Guidelines

This C sentence describes the behavior when a conversion adds qualifiers to a pointer type. The standard does not contain a sentence that specifically discusses the behavior when conversions remove qualifiers from a pointer type. This issue is discussed elsewhere.

pointer 758 converted to pointer to different object or type

---

quali-fied/unqualified pointer compare equal

the values stored in the original and converted pointers shall compare equal. 747

Commentary

This is a requirement on the implementation. Qualifiers provide additional information on the properties of objects, they do not affect the representation of the pointer used.

pointer 559 to quali-fied/unqualified types

C++

3.9.2p3 *Pointers to cv-qualified and cv-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.9).*

By specifying layout-compatible types, not the same type, the C++ Standard restricts the freedom of implementations more than C99 does.

Common Implementations

Qualifiers provide information that enables translators to optimize, or not, programs. It is very unlikely that the conversion operation itself will result in any machine code being generated. The likely impact (quality of machine code generated by a translator) will be on subsequent accesses using the pointer value, or the pointed-to object.

---

null pointer constant

An integer constant expression with the value 0, or such an expression cast to type **void** `*`, is called a *null pointer constant*.[55] 748

Commentary

This defines the term *null pointer constant*, which is rarely used in its full form by developers. The shortened term *null pointer* is often used to cover this special case (which technically it does do). 749 null pointer 1333 constant null pointer constant

Note that a constant expression is required— a value known at translation time. A value of zero computed during program execution is not a null pointer constant. The reason for this distinction is that the token `0` (and the sequence of tokens that includes a cast operation) is purely an external representation, in the source code, of something called a null pointer constant. The internal, execution-time value can be any sequence of bits. The integer constant expression `(1-1)`, for example, has the value `0` and could be used to denote the null pointer constant (the response to DR #261 confirmed this interpretation). However, while the value of the expression `(x-x)`, where `x` is an initialized integer object, might be known at translation time it does not denote a null pointer constant.

Developers (and implementations) often treat any cast of the value `0` to a pointer type as a null pointer constant. For instance, the expression `(char *)0` is often used to represent a null pointer constant in source code. While it is a null pointer, it is not the null pointer constant. Such usage is often a hangover from the prestandard days, before the type **`void`** was introduced.

C++

A null pointer constant is an integral constant expression (5.19) rvalue of integer type that evaluates to zero. 4.10p1

The C++ Standard only supports the use of an integer constant expression with value `0`, as a null pointer constant. A program that explicitly uses the pointer cast form need not be conforming C++; it depends on the context in which it occurs. Use of the implementation-provided `NULL` macro avoids this compatibility problem by leaving it up to the implementation to use the appropriate value.

The C++ Standard specifies the restriction that a null pointer constant can only be created at translation time.

64) Converting an integral constant expression (5.19) with value zero always yields a null pointer (4.10), but converting other expressions that happen to have value zero need not yield a null pointer. Footnote 64

Other Languages

Many languages use a reserved word, or keyword (**`nil`** is commonly seen), to represent the concept and value of the null pointer.

Common Implementations

Many implementations use an execution-time representation of all bits zero as the value of the null pointer constant. Implementations for processors that use a segmented memory architecture have a number of choices for the representation of the null pointer. The Inmos Transputer[622] has a signed address space with zero in the middle and uses the value 0x80000000 as its execution-time representation; the IBM/390 running CICS also used this value. Non-zero values were also used on Prime and Honeywell/Multics.[179] 590 pointer segmented architecture

Coding Guidelines

Technically there is a distinction between a null pointer constant and a null pointer. In practice implementations rarely make a distinction. It is debatable whether there is a worthwhile benefit in trying to educate developers to distinguish between the two terms. 749 null pointer

As the following two points show attempting to chose between using `0` and `(void *)0` involves trading off many costs and benefits:

- The literal `0` is usually thought about in arithmetic, rather than pointer, terms. A cognitive switch is needed to think of it as a null pointer constant. Casting the integer constant to pointer to **`void`** may remove the need for a cognitive switch, but without a lot of practice (to automate the process) recognizing the token sequence `(void *)0` will need some amount of conscious effort. 0 cognitive switch

- When searching source using automated tools (e.g., `grep`), matches against the literal `0` will not always denote the null pointer constant. Matches against `(void *)0` will always denote the null pointer constant; however, there are alternative character sequences denoting the same quantity (e.g., `(void*)0`).

However, there is a third alternative. The `NULL` macro is defined in a number of standard library headers. Use of this macro has the advantages that the name is suggestive of its purpose; it simplifies searches (although the same sequence of characters can occur in other names) and source remains C++ compatible. It is likely that one of the standard library headers, defining it, has already been **`#include`**d by a translation unit. If not, the cost of adding a **`#include`** preprocessor directive is minimal.

Cg 748.1

The null pointer constant shall only be represented in the visible form of source code by the `NULL` macro.

Casting the value of an object having an integer type to a pointer type makes use of undefined behavior. While the null pointer constant is often represented during program execution, by all bits zero, there is no requirement that any particular representation be used.

Example

```
1  #include <stdio.h>
2
3  extern int glob = 3;
4
5  char *p_c_1 = 0;
6  char *p_c_2 = (void *)0;
7  char *p_c_3 = 9 - 8 - 1;
8  char *p_c_4 = (1 == 2) && (3 == 4);
9  char *p_c_5 = NULL;
10
11 void f(void)
12 {
13 if (NULL != (void *)(glob - glob))
14    printf("Surprising, but possible\n");
15 }
```

---

null pointer

If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function. 749

Commentary

This is a requirement on the implementation. The null pointer constant is often the value used to indicate that a pointer is not pointing at an object; for instance, on a linked list or other data structure, that there are no more objects on the list. Similarly, the null pointer constant is used to indicate that no function is referred to by a pointer-to function.

The `(void *)0` form of representing the null pointer constant already has a pointer type.

C++

4.10p1 *A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of pointer to object or pointer to function type.*

A null pointer constant (4.10) can be converted to a pointer to member type; the result is the null member pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant.

Presumably *distinguishable* means that the pointers will compare unequal.

Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array. 5.10p1

From which we can deduce that a null pointer constant cannot point one past the end of an object either.

Other Languages

Languages that have a construct similar to the null pointer constant usually allow it to be converted to different pointer types (in Pascal it is a reserved word, **`nil`**).

Common Implementations

All bits zero is a convenient execution-time representation of the null pointer constant for many implementations because it is invariably the lowest address in storage. (The INMOS Transputer[622] had a signed address space, which placed zero in the middle.) Although there may be program bootstrap information at this location, it is unlikely that any objects or functions will be placed here. Many operating systems leave this storage location unused because experience has shown that program faults sometimes cause values to be written into the location specified by the null pointer constant (the more developer-oriented environments try to raise an exception when that location is accessed).

Another implementation technique, when the host environment does not include address zero as part of a processes address space, is to create an object (sometimes called `__null`) as part of the standard library. All references to the null pointer constant refer to this object, whose address will compare unequal to any other object or function.

---

750 Conversion of a null pointer to another pointer type yields a null pointer of that type.

null pointer conversion yields null pointer

Commentary

This is a requirement on the implementation. It is the only situation where the standard guarantees that a pointer to any type may be converted to a pointer to a completely different type, and will deliver a defined result. (Conversion to pointer to **`void`** requires the pointer to be converted back to the original type before the value is defined.)

743 pointer to void converted to/from
744 pointer converted to pointer to void

C90

The C90 Standard was reworded to clarify the intent by the response to DR #158.

Common Implementations

In most implementations this conversion leaves the representation of the null pointer unmodified and is essentially a no-op (at execution time). However, on a segmented architecture, the situation is not always so simple. For instance, in an implementation that supports both near and far pointers, a number of possible representation decisions may be made, including:

590 pointer segmented architecture

- The *near* null pointer constant might have offset zero, while the *far* null pointer constant might use an offset of zero and a segment value equal to the segment value used for all near pointers (which is likely to be nonzero because host environments tend to put their own code and data at low storage locations). When the integer constant zero is converted to a *far* null pointer constant representation, translators can generate code to create the appropriate segment offset. When objects having an integer type are converted to pointers, implementations have to decide whether they are going to treat the value as a meaningless bit pattern, or whether they are going to check for and special-case the value zero. A similar decision on treating pointer values as bit patterns or checking for special values has to be made when generating code to convert pointers to integers.

- The *near* null pointer constant might have offset zero, while the *far* null pointer constant might have both segment and offset values of zero. In this case the representation is always all bits zero; however, there are two addresses in storage that are treated as being the null pointer constant. When two pointers that refer to the same object are subtracted, the result is zero; developers invariably extend this behavior to include the subtraction of two null pointer constants (although this behavior is not guaranteed by the standard because the null pointer constant does not refer to an object). Because the null pointer constant may be represented using two different addresses in storage, implementation either has to generate code to detect when two null pointer constants are being subtracted or be willing to fail to meet developer expectations in some cases.

Debugging implementations that make use of a processor's automatic hardware checking (if available) of address accesses may have a separate representation (e.g., offset zero and a unique segment value) for every null pointer constant in the source. (This enables runtime checks to trace back dereferences of the null pointer constant to the point in the source that created the constant.)

---

null pointer compare equal

Any two null pointers shall compare equal. 751

Commentary

This is a requirement on the implementation. They compare equal using the equality operator (they are not required to have the same, bit-for-bit, representations) irrespective of their original pointer type. (There may be equality operator constraints that effectively require one or the other of the operands to be cast before the equality operation is performed.)

equality operators constraints 1213

C++

4.10p1 *Two null pointer values of the same type shall compare equal.*

4.11p1 *Two null member pointer values of the same type shall compare equal.*

The C wording does not restrict the null pointers from being the same type.

4.10p3 *The null pointer value is converted to the null pointer value of the destination type.*

This handles pointers to class type. The other cases are handled in 5.2.7p4, 5.2.9p8, 5.2.10p8, and 5.2.11p6.

Other Languages

Null pointers are usually considered a special value and compare equal, independent of the pointer type.

---

integer permission to convert to pointer

An integer may be converted to any pointer type. 752

Commentary

This is the one situation where the standard allows a basic type to be converted to a derived type. Prior to the C90 Standard, a pointer value had to be input as an integer value that was then cast to a pointer type. The `%p` qualifier was added to the `scanf` library function to overcome this problem.

Other Languages

Many other languages permit some form of integer-to-pointer type conversion. Although supporting this kind of conversion is frowned on by supporters of strongly typed languages, practical issues (it is essential in some applications) mean that only the more academic languages provide no such conversion path. Java does not permit an integer to be converted to a reference type. Such a conversion would allow the security in the Java execution-time system to be circumvented.

Common Implementations

Most implementations do not generate any machine code for this conversion. The bits in the integer value are simply interpreted as having a pointer value.

Coding Guidelines

The standard may allow this conversion to be made, but why would a developer ever want to do it? Accessing specific (at known addresses) storage locations might be one reason. Some translators include an extension that enables developers to specify an object's storage location in its definition. But this extension is only available in a few translators. Casting an integer value to a pointer delivers predictable results on a wider range of translators. The dangers of converting integer values to pointers tend to be talked about more often than the conversion is performed in practice. When such a conversion is used, it is often essential.

1348 object specifying address

In some development environments (e.g., safety-critical) this conversion is sometimes considered sufficiently dangerous that it is often banned outright. However, while use of the construct might be banned on paper, experience suggests that such usage still occurs. The justifications given to support these usages, where it is necessary to access specific storage locations, can involve lots of impressive-sounding technical terms. Such obfuscation serves no purpose. Clear rationale is in everybody's best interests. Such conversions make use of representation information and, as such, are covered by a guideline recommendation. The following wording is given as a possible deviation to this representation usage guideline recommendation:

569.1 representation information using

Dev 569.1 Values whose integer type has a rank that is less than or equal to that of the type `intptr_t` may be converted to a pointer type.

Dev 569.1 Integer values may be converted to a pointer type provided all such conversions are commented in the source code and accompanied by a rationale document.

Example

```
1   char *p_c = (char *)43;
```

---

753 Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.[56)]

integer-to-pointer implementation-defined

Commentary

This is a complete list of all the issues that need to be considered when relying on such a conversion. The integer constant zero is a special case. C does not provide any mechanisms for checking that a pointer is well-formed in the sense of being correctly aligned or pointing at an object whose lifetime has not terminated.

748 null pointer constant

There is a long-standing developer assumption that a pointer type occupies the same number of value bits as the type **long**. The growing availability of processors using 64-bit addressing support for type **long long** (which often means a 32-bit representation is used for type **long**) means that uncertainty over the minimum rank of the integer type needed to represent a pointer value is likely to grow. The introduction of the typedef name `intptr_t` is intended to solve this problem.

C++

The C++ Standard specifies the following behavior for the **reinterpret_cast**, which is equivalent to the C cast operator in many contexts.

> *A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.*

5.2.10p5

The C++ Standard provides a guarantee— a round path conversion via an integer type of sufficient size (provided one exists) delivers the original value. Source developed using a C++ translator may contain constructs whose behavior is implementation-defined in C.

The C++ Standard does not discuss trap representations for anything other than floating-point types.

Other Languages

The above issues are not specific to C, but generic to languages designed to run on a wide range of processors. Anything that could go wrong in one language could go equally wrong in another.

Common Implementations

Most implementations take the pattern of bits in the integer value and interpret them as a pointer value. They do not perform any additional processing on the bit pattern (address representations used by implementations are discussed elsewhere).

byte 54
address unique

Just because the equality `sizeof(char *) == sizeof(int *)` holds does not mean that the same representations are used. For instance, one Data General MV/Eclipse C compiler used different representations, although the sizes of the pointer types were the same. The IAR PICmicro compiler[612] provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size. On such a host, the developer needs to be aware of the storage bank referred to by a particular pointer. Converting an integer type to a pointer-to type using the ILE C translator for the IBM AS/400[617] causes the right four bytes of the 16-byte pointer to hold the integer value. This pointer value cannot be dereferenced.

Coding Guidelines

representation information using 569.1

If the guideline recommendation dealing with use of representation information is followed, then this conversion will not occur. If there are deviations to this guideline recommendation, it is the developer's responsibility to ensure that the expected result is obtained. Possible deviations and situations to watch out for are not sufficiently common to enable deviations to be specified here.

---

pointer permission to convert to integer

Any pointer type may be converted to an integer type. 754

Commentary

integer 752
permission to convert to pointer

The inverse operation to the one described in a previous C sentence. Prior to the introduction of the C Standard, C90, pointer values had to be converted to integers before they could be output. The `%p` conversion specifier was added to the `printf` family of functions to overcome this problem. Pointer types are sometimes converted to integers because developers want to perform bit manipulation on the value, often followed by a conversion back to the original pointer type. Application areas that perform this kind of manipulation include garbage collectors, tagged pointers (where unused address bits are used to hold information on the pointed-to object), and other storage management functions.

C++

5.2.10p4 *A pointer can be explicitly converted to any integral type large enough to hold it.*

The C++ wording is more restrictive than C, which has no requirement that the integer type be large enough to hold the pointer.

integer-to-pointer implementation-defined 753

While the specification of the conversion behaviors differ between C++ and C (undefined vs. implementation-defined, respectively), differences in the processor architecture is likely to play a larger role in the value of the converted result.

Other Languages

There are fewer practical reasons for wanting to convert a pointer-to integer type than the reverse conversion, and languages are less likely to support this kind of conversion. Java does not permit a reference type to be converted to an integer. Such a conversion would allow the security in the Java execution-time system to be circumvented.

Common Implementations

Most implementations do not generate any machine code for this conversion. The bits in the integer value are simply interpreted as having a pointer value.

Coding Guidelines

Pointer-to-integer conversions differ from integer-to-pointer conversions in that there are likely to be fewer cases where a developer would need to perform them. The C language supports pointer arithmetic, so it is not necessary to convert the value to an integer type before incrementing or decrementing it. One use (perhaps the only real one) of such a conversion is the need to perform some nonarithmetic operation, for instance a bitwise operation, on the underlying representation. If bit manipulation is to be carried out, then type punning (e.g., a union type) is an alternative implementation strategy.

531 type punning union

Such conversions make use of representation information and as such are covered by a guideline recommendation. The following wording is given as a possible deviation to this representation usage guideline recommendation (it mirrors those suggested for integer-to-pointer conversions):

569.1 representation information using
752 integer permission to convert to pointer

Dev 569.1 A value having a pointer type may only be converted to an integer type whose rank is greater than or equal to that of the type `intptr_t`.

Dev 569.1 A value having a pointer type may only be converted to an integer type provided all such conversions are commented in the source code and accompanied by a rationale document.

Example

```
1  #include <stdint.h>
2
3  extern char *c_p;
4
5  void f(void)
6  {
7  unsigned char l_uc  = (unsigned char)c_p;
8  int           l_i   = (int)          c_p;
9  intptr_t      l_ip  = (intptr_t)     c_p;
10 uintptr_t     l_uip = (uintptr_t)    c_p;
11 long long     l_ll  = (long long)    c_p;
12 }
```

Usage

Usage information on pointer conversions is given elsewhere (see Table 758.1 and Figure 1134.1).

---

755 Except as previously specified, the result is implementation-defined.

Commentary

There is both implementation-defined and undefined behavior involved here. By specifying implementation-defined behavior, if the result can be represented, the Committee is requiring that the developer have access to information on the converted representation (in the accompanying documentation).

756 pointer conversion undefined behavior
42 implementation-defined behavior

Other Languages

Like integer-to-pointer conversions the issues tend to be generic to all languages. In many cases language specification tends to be closer to the C term *undefined behavior* rather than implementation-defined behavior.

753 integer-to-pointer implementation-defined

Common Implementations

Like integer-to-pointer conversions, most implementations simply take the value representation of a pointer and interpret the bits (usually the least significant ones if the integer type contains fewer value bits) as the appropriate integer type. Choosing a mapping such that the round trip of pointer-to-integer conversion,

integer-753 to-pointer implementation-defined

followed by integer-to-pointer conversion returns a pointer that refers to the same storage location is an objective for some implementations.

Some pointer representations contain status information, such as supervisor bits, as well as storage location information. The extent to which this information is included in the integer value can depend on the number of bits available in the value representation. Converting a pointer-to function type to an integer type using the ILE C translator for the IBM AS/400[617] always produces the result 0.

---

pointer conversion undefined behavior

If the result cannot be represented in the integer type, the behavior is undefined. 756

Commentary

The first requirement is that the integer type have enough bits to enable it to represent all of the information present in the pointer value. There then needs to be a mapping from pointer type values to the integer type.

C90

If the space provided is not long enough, the behavior is undefined.

The C99 specification has moved away from basing the specification on storage to a more general one based on representation.

C++

The C++ Standard does not explicitly specify any behavior when the result cannot be represented in the integer type. (The wording in 5.2.10p4 applies to "any integral type large enough to hold it.")

Common Implementations

Most implementations exhibit no special behavior if the result cannot be represented. In most cases a selection of bits (usually the least significant) from the pointer value is returned as the result.

The result need not be in the range of values of any integer type. 757

Commentary

This statement means that there is no requirement on the implementation to provide either an integer type capable of representing all the information in a pointer value, or a mapping if such an integer type is available

C90

The C90 requirement was based on sufficient bits being available, not representable ranges.

C++

There is no equivalent permission given in the C++ Standard.

Common Implementations

Implementation vendors invariably do their best to ensure that such a mapping is supported by their implementations. The IBM AS/400[616] uses 16 bytes to represent a pointer value (the option `datamodel` can be used to change this to 8 bytes), much larger than can be represented in any of the integer types available on that host.

Coding Guidelines

While the typedef names `intptr_t` and `uintptr_t` (specified in the library subsection) provide a mechanism for portably representing pointer values in an integer type, support for them is optional. An implementation is not required to provide definitions for them in the **<stdint.h>** header.

---

pointer converted to pointer to different object or type

pointer 743 to void converted to/from

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. 758

Commentary

This is a more general case of the permission given for pointer to **void** conversions.

C++

The C++ Standard states this in 5.2.9p5, 5.2.9p8, 5.2.10p7 (where the wording is very similar to the C wording), and 5.2.11p10.

Other Languages

Many languages that support pointer types and a cast operator also support some form of conversion between different pointers.

Coding Guidelines

Experience suggests that there are two main reasons for developers wanting to convert a pointer-to object type to point at different object types:

1. Pointer-to object types are often converted to pointer-to character type. A pointer to **unsigned char** is commonly used as a method of accessing all of the bytes in an object's representation (e.g., so they can be read or written to a different object, file, or communications link). The standard specifies requirements on the representation of the type **unsigned char** in order to support this common usage. The definition of effective type also recognizes this usage,

571 unsigned char pure binary
948 effective type

2. Pointers to different structure types are sometimes cast to each other's types. This usage is discussed in detail elsewhere.

1037 union special guarantee

Pointer conversions themselves are rarely a cause of faults; it is the subsequent accesses to the referenced storage that is the source of the problems. Pointer conversion can be looked on as a method of accessing objects using different representations. As such, it is implicitly making use of representation information, something which is already covered by a guideline recommendation. Recognizing that developers do sometimes need to make use of representation information, the following wording is given as a possible deviation to this representation usage guideline recommendation:

569.1 representation information using

Dev 569.1 A pointer-to character type may be converted to a pointer-to another type having all of the qualifiers of the original pointed-to type.

Dev 569.1 A pointer may be converted to a pointer-to character type having all of the qualifiers of the original pointed-to type.

Dev 569.1 A pointer-to structure type may be converted to a different pointer-to structure type provided they share a common initial sequence.

Example

```
1  extern int *p_i;
2  extern const float *p_f;
3  extern unsigned char *p_uc;
4
5  void f(void)
6  {
7  p_uc = (unsigned char *)p_i;
8  /* ... */
9  p_uc = (unsigned char *)p_uc;
10 p_uc = (unsigned char const *)p_uc;
11 }
```

Table 758.1: Occurrence of implicit conversions that involve pointer types (as a percentage of all implicit conversions that involve pointer types). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
`( struct * )`	`int`	44.0	`( void * )`	`int`	4.2
(function *)	`int`	18.4	`( unsigned char * )`	`int`	3.4
`( char * )`	`int`	7.9	(ptr-to *)	`int`	2.0
`( const char * )`	`int`	6.9	`( int * )`	`int`	1.9
`( union * )`	`int`	5.5	`( long * )`	`int`	1.1
(other-types *)	other-types *	4.7			

If the resulting pointer is not correctly aligned[57)] for the pointed-to type, the behavior is undefined. 759

Commentary

alignment 39

It is the pointed-to object whose alignment is being discussed here. Object types may have different alignment requirements, although the standard guarantees certain alignments. It is guaranteed that converting a pointer value pointer to **`void`** and back to the original type will produce a result that compares equal to the original value.

pointer 744 converted to pointer to void

alignment 39

What does the term *correctly* mean here? Both C implementations and processors may specify alignment requirements, but the C Standard deals with implementations (which usually take processor requirements into account). It needs to be the implementation's alignment requirements that are considered correct, if they are met. Different processors handle misaligned accesses to storage in different ways. The overhead of checking each access, via a pointer value, before it occurred would be excessive. The Committee traded off performance for undefined behavior.

C++

5.2.10p7 *Except that converting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.*

The unspecified behavior occurs if the pointer is not cast back to its original type, or the relative alignments are stricter.

Source developed using a C++ translator may contain a conversion of a pointer value that makes use of unspecified behavior, but causes undefined behavior when processed by a C translator.

Other Languages

storage layout 1354

Alignment is an issue resulting from processor storage-layout requirements. As such the resulting undefined behavior could almost be said to be language-independent.

Common Implementations

Most implementations treat a pointer-to-pointer conversion as a no-op in the generated machine code; that is, no conversion code is actually generated. The original bit pattern is returned. The effects of alignment usually become apparent later when the object referenced by the converted pointer value is accessed. An incorrectly aligned pointer value can cause a variety of different execution-time behaviors, including raising a signal.

The Unisys A Series[1390] uses byte addresses to represent pointer-to character types and word addresses (six bytes per word) to represent pointers to other integer types. Converting a pointer to **`char`** to a pointer to **`int`** requires that the pointer value be divided by six. Unless the **`char`** object pointed to is on a 6-byte address boundary, the converted pointer will no longer point at it.

Coding Guidelines

Converting pointer type requires developers to consider a representation issue other than that of the pointed-to types alignment. The guideline recommendation dealing with use of representation information is applicable here.

39 alignment
569.1 representation information using

What if a deviation from the guideline recommendation dealing with use of representation information is made? In source code that converts values having pointer types, alignment-related issues are likely to be encountered quickly during program testing. However, pointer conversions that are correctly aligned on one processor may not be correctly aligned for another processor. (This problem is often encountered when porting a program from an Intel x86-based host, few alignment restrictions, to a RISC-based host, which usually has different alignment requirements for the different integer types.) Alignment of types, as well as representation, thus needs to be considered when creating a deviation for conversion of pointer types.

---

760 Otherwise, when converted back again, the result shall compare equal to the original pointer.

pointer converted back to pointer
745 converted via pointer to void compare equal

Commentary

This is a requirement on the implementation. It mirrors the requirement given for pointer to **void**, except that in this case there is an issue associated with the relative alignments of the two pointer types.

C++

Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.

5.2.10p7

The C++ Standard does not specify what *original pointer value* means (e.g., it could be interpreted as bit-for-bit equality, or simply that the two values compare equal).

Common Implementations

For most implementations the converted and original values are bit for bit identical, even when the target processor uses a segmented architecture. (On such processors, it is usually only the result of arithmetic operations that need to be checked for being in canonical form.)

590 pointer segmented architecture

---

761 When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object.

pointer converted to pointer to character object lowest addressed byte
570 object contiguous sequence of bytes

Commentary

This is a requirement on the implementation. By specifying an address, the standard makes it possible to predictably walk a pointer over the entire storage space allocated to the object (using **sizeof** to determine how many bytes it contains). The standard could equally have chosen the highest address. But common developer thinking is for addresses to refer to the beginning of an object, which is the lowest address when storage starts at zero and increases. The vast majority of processors have an address space that starts at zero (the INMOS Transputer[622] has a signed address space). Instructions for loading scalars from a given address invariably assume that successive bytes are at greater addresses.

The lowest address of an object is not required to correspond to the least significant byte of that object, if it represents a scalar type.

C90

The C90 Standard does not explicitly specify this requirement.

C++

4.10p2

The result of converting a "pointer to `cv` *T" to a "pointer to* `cv void`*" points to the start of the storage location where the object of type T resides, . . .*

However, the C wording is for pointer-to character type, not pointer to **`void`**.

3.9.2p4 *A cv-qualified or cv-unqualified (3.9.3)* `void*` *shall have the same representation and alignment requirements as a cv-qualified or cv-unqualified* `char*`.

5.2.10p7 *A pointer to an object can be explicitly converted to a pointer to an object of different type[65]. Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.*

The C++ Standard does not require the result of the conversion to be a pointer to the lowest addressed byte of the object. However, it is very likely that C++ implementations will meet the C requirement.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

representation information using 569.1

Needing to know that an address refers to the lowest addressed byte of an object suggests that operations are performed at this level of implementation detail (which violates the guideline recommendation dealing with use of representation information). However, there are situations where the specification of different functionality, in the standard, is interconnected. For instance, the specification of the `memcpy` library function needs to know whether the addresses passed to it points at the lowest or highest byte.

Dev 569.1

A program may depend on the lowest address of a byte being returned by the address-of operator (after conversion to pointer-to character type) when the result is either passed as an argument to a standard library function defined to accept such a value, or assigned to another object that is used for that purpose.

---

footnote 55

55) The macro **`NULL`** is defined in **`<stddef.h>`** (and other headers) as a null pointer constant; see 7.17. 762

Commentary

It is also defined in the headers **`<locale.h>`**, **`<stdio.h>`**, **`<stdlib.h>`**, **`<string.h>`**, **`<time.h>`**, and **`<wchar.h>`**.

Other Languages

Many languages defined a reserved word to represent the null pointer constant. The word **`nil`** is often used.

Coding Guidelines

null pointer constant 748

Some existing source code provides its own definition of this macro rather than **`#include`**ing the appropriate header. Given that the value of the null pointer constant is defined by the standard this usage would appear to be harmless.

---

footnote 56 pointer/integer consistent mapping

56) The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment. 763

Commentary

Allowing conversions between pointers and integers is of no practical use unless implementations are willing to go along with developers' expectations of being able to convert pointers in some meaningful way. One, formally validated, C90 implementation[681] represents pointers as an index into an array (the array holding the base address of the actual object and the offset within it). In this implementation a pointer-to integer conversion yielded the value of the array index, not the actual address of the object. While presence of this footnote does not prevent such an implementation from conforming to C99, its behavior might limit the number of product sales made.

C++

[Note: it is intended to be unsurprising to those who know the addressing structure of the underlying machine.] 5.2.10p4

Is an unsurprising mapping the same as a consistent one? Perhaps an unsurprising mapping is what C does. :-)

Other Languages

While languages do not get involved in this level of detail, their implementations often pander to developer expectations.

Common Implementations

The intended execution environments addressing structure is the one visible to an executing program. The extent to which the underlying hardware addressing is visible to programs varies between target processors. A processor may have a linear address space at the physical level, but at the logical level the addressing structure could be segmented. It is likely to be in vendors' commercial interest for their implementations to support some form of meaningful mapping. 590 pointer segmented architecture

Some of the issues involved in representing the null point constant in a consistent manner are discussed elsewhere. 748 null pointer constant

Coding Guidelines

This footnote expresses the assumption that C developers might want to convert between integer and pointer types in some meaningful way. Such usage is making use of representation information and is covered by a guideline recommendation. 569.1 representation information using

---

764 57) In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C. footnote 57

Commentary

Your author knows of no implementations where the concept *correctly aligned* is not transitive.

C90

This observation was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not point out that alignment is a transitive property.

---

765 Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object. object point at each bytes of

Commentary

Here the standard is specifying the second of the two requirements needed to allow strictly conforming C code to handle objects as a sequence of bytes, via a pointer-to character type (the other is that the address of

byte 54 address unique
sizeof char 1124 defined to be 1
object 570 contiguous sequence of bytes
array 526 contiguously allocated set of objects
value 573 copied using unsigned char
pointer 1169 one past end of object

individual bytes is unique). The increment value here being `sizeof(unsigned char)`, which is defined to be 1.

The standard specifies elsewhere that the bytes making up an object are contiguously allocated. There is also a guarantee to be able to increment the result one past the end of the object.

C90

The C90 Standard does not explicitly specify this requirement.

C++

The equivalent C++ requirement is only guaranteed to apply to pointer to **`void`** and it is not possible to perform arithmetic on this pointer type. However, in practice C++ implementations are likely to meet this C requirement.

Other Languages

Most languages do not support the concept of converting a pointer-to character type for the purpose of accessing the bytes of the pointed-to object. However, this concept is sometimes used by developers in other languages (they have to make use of particular implementation details that are not given any language standard's blessing).

Coding Guidelines

Pointing at the individual bytes making up an object is one of the steps involved in making use of representation information. The applicable coding guideline discussion is the one dealing with the overall general issue of using representation information complete process, not the individual steps involved.

types 569 representation

---

pointer to function converted

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; 766

Commentary

pointer 758 converted to pointer to different object or type

This is a requirement on the implementation. It mimics the one specified for pointer-to object conversions. The standard is silent on the issue of converting a pointer-to function type to a pointer-to object type. As such the behavior is implicitly undefined.

Other Languages

Few languages support pointers to functions. Some languages allow objects to have a function type, they can be assigned references to functions but are not treated as pointers. Those languages that do support some form of indirect function call via objects do not always support casting of function types.

Common Implementations

Most implementations use the same representation for pointers to objects and pointers to functions (the pointer value being the address of the function in the program's address space). The C compiler for the Unisys e-@ction Application Development Solutions[1391] returns the value 32 when **`sizeof`** is applied to a pointer-to function and 8 when it is applied to a pointer-to object.

call function 768 via converted pointer

Although pointers to different function types usually have the same representation and alignment requirements, the same cannot be said for their argument-passing conventions. Some implementations represent a pointer-to function type as an index into a table. This approach is sometimes used to provide a form of memory management, in software, when this is not provided by the target host hardware. Accessing functions via an index into a table, rather than jumping directly to their address in memory, provides the hook for a memory manager to check if the machine code for a function needs to be swapped into memory because it has been called (and potentially another function machine code swapped out).

The IAR PICmicro compiler[612] offers four different storage banks that pointers to functions can refer to. Part of the information needed to deduce the address of a function is the storage bank that contains it (the translator obtains this information from the declaration of each pointer type). The value of a pointer-to function may be assigned to another object having a pointer-to function type, but information on the bank it refers to is not part of the value representation. If two pointers to function refer to different storage banks, it

is not possible to call a function referred to by one via the other. This compiler also provides a mechanism for specifying which storage bank is to be used to pass the arguments in a call to a particular function.

Coding Guidelines

Pointers to functions are usually converted to another pointer type for the same reasons as pointer-to object types are converted to pointer to **void**; the desire for a single function accepting arguments that have different pointer-to function types. Manipulation of functions as pointer values is not commonly seen in code and developers are likely to have had little practice in reading or writing the types involved. The abstract declarator that is the pointer-to function type used in the conversion tends to be more complex than most other types, comprising both the return type and the parameter information. This type also needs to be read from the inside out, not left-to-right.

744 pointer converted to pointer to void

1624 abstract declarator syntax

Is there a pointer-to function type equivalent to the pointer to **void** type for pointer-to function conversions? The keyword **void** is an explicit indicator that no information is available. In the case of pointer-to function conversions either the return type, or the parameter type, or both could be unknown.

Example

```
1   extern int f_1(int (*)(void));
2   extern int f_2(void (*)(int, char));
3
4   extern int g_1a(int);
5   extern int g_1b(long);
6   extern long g_2a(int, char);
7   extern float g_2b(int, char);
8
9   void h(void)
10  {
11  f_1((int (*)(void))g_1a);
12  f_1((int (*)(void))g_1b);
13
14  f_2((void (*)(int, char))g_2a);
15  f_2((void (*)(int, char))g_2b);
16  }
```

---

767 the result shall compare equal to the original pointer.

Commentary

This is a requirement on the implementation. This is the only guarantee on the properties of pointer-to function conversion. It mimics that given for pointer-to object conversions.

760 pointer converted back to pointer

Other Languages

Those languages that do support some form of indirect function call, via values held in objects, often support equality comparisons on the values of such objects, even if they do not support conversions on these types.

Example

```
1   extern int f(void);
2
3   void g(void)
4   {
5   if ((int (*)(void))(int (*)(float, long long))f != f)
6      ; /* Complain. */
7   }
```

call function via converted pointer

If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined. 768

Commentary

In practice the representation of pointers to functions is rarely dependent on the type of the function (although it may be different from the representation used for pointers to objects). The practical issue is one of argument-passing. It is common practice for translators to use different argument passing conventions for functions declared with different parameters. (It may depend on the number and types of the parameters, or whether a prototype is visible or not.) For instance, an implementation may pass the first two arguments in registers if they have integer type; but on the stack if they have any other type. The code generated by a translator for the function definition and calls to it are aware of this convention. If a pointer to a function whose first parameter has a structure type (whose first member is an integer) is cast to a pointer to a function whose first parameter is the same integer type (followed by an ellipsis), a call via the converted pointer will not pass the parameters in the form expected by the definition (the first member of the structure will not be passed in a register, but on the stack with all the other members).

C++

5.2.10p6 *The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined.*

compatible type if 631

C++ requires the parameter and return types to be the same, while C only requires that these types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

Common Implementations

In the majority of implementations a function call causes a jump to a given address. The execution of the machine instructions following that address will cause accesses to the storage locations or registers, where the argument values are expected to have been assigned to. Thus the common undefined behavior will be to access these locations irrespective of whether anything has been assigned to them (potentially leading to other undefined behaviors).

Coding Guidelines

Conversions involving pointer-to function types occur for a variety of reasons. For instance, use of callback functions, executing machine code that is known to exist at a given storage location (by converting an integer value to pointer-to function), an array of pointer-to function (an indirect call via an array index being deemed more efficient, in time and/or space, than a **switch** statement). While indirect function calls are generally rare, they do occur relatively often in certain kinds of applications (e.g., callbacks in GUI interface code, dispatchers in realtime controllers).

Example

```
1  extern void f(void);
2
3  extern int (*p_f_1)(void);
4  extern int (*p_f_2)(int);
5  extern int (*p_f_3)(char, float);
6  extern int (*p_f_4)();
7
8  int (*p_f_10)(void) = (int (*)(void))f;
9  long (*p_f_11)(int) = (long (*)(int))f;
10 int (*p_f_12)(char, float) = (int (*)(char, float))f;
11 int (*p_f_13)() = (int (*)())f;
```

```
12
13  void f(void)
14  {
15  ((int (*)())p_f_1)();
16  ((int (*)(int))p_f_1)(2);
17  ((void (*)(void))p_f_1)();
18  ((int (*)())p_f_2)();
19  ((int (*)(float, char))p_f_3)(1.2, 'q');
20  ((int (*)(void))p_f_4)();
21  }
```

769 **Forward references:** cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.18.1.4), simple assignment (6.5.16.1).

6.4 Lexical elements

token syntax preprocessing token syntax

770

token:
keyword
identifier
constant
string-literal
punctuator

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

Commentary

Tokens (preprocessor or otherwise) are the atoms from which the substance of programs are built. Preprocessing tokens are created in translation phase 3 and are turned into tokens in translation phase 7.

124 translation phase 3
136 translation phase

All characters in the basic source character set can be used to form some kind of preprocessing token that is defined by the standard. When creating preprocessing tokens the first non-white-space character is sufficient, in all but one case, to determine the kind of preprocessing token being created.

punctuator 912 syntax
header name 918 syntax

The two characters, double-quote and single-quote, must occur with their respective matching character if they are to form a defined preprocessor-token. The special case of them occurring singly, which matches against "non-white-space character that cannot be one of the above," is dealt with below. The other cases that match against *non-white-space character that cannot be one of the above* involve characters that are outside of the basic source character set. A program containing such extended characters need not result in a constraint violation, provided the implementation supports such characters. For instance, they could be stringized prior to translation phase 7, or they could be part of a preprocessor-token sequence being skipped as part of a conditional compilation directive.

character 776 ' or " matches
basic source character set 221
extended characters 215
preprocessing token 771 shall have lexical form
1950 operator

The header-name preprocessing token is context-dependent and only occurs in the **#include** preprocessing directive. It never occurs after translation phase 4.

#include 1897 h-char-sequence

C90

The *non-terminal* operator was included as both a *token* and *preprocessing-token* in the C90 Standard. Tokens that were operators in C90 have been added to the list of punctuators in C99.

C++

C++ maintains the C90 distinction between operators and punctuators. C++ also classifies what C calls a constant as a literal, a string-literal as a literal and a C character-constant is known as a character-literal.

Other Languages

Few other language definitions include a preprocessor. The PL/1 preprocessor includes syntax that supports some statements having the same syntax as the language to be executed during translation.

Some languages (e.g., PL/1) do not distinguish between keywords and identifiers. The context in which a name occurs is used to select the usage to which it is put. Other languages (e.g., Algol 60) use, conceptually, one character set for keywords and another for other tokens. In practice only one character set is available. In books and papers the convention of printing keywords in bold was adopted. A variety of conventions were used for writing Algol keywords in source, including using an underline facility in the character encodings, using matching single-quote characters, or simply all uppercase letters.

Common Implementations

The handling of "each non-white-space character that cannot be one of the above" varies between implementations. In most cases an occurrence of such a preprocessing token leads to a syntax or constraint violation.

Coding Guidelines

Most developers are not aware of that preprocessing-tokens exist. They think in terms of a single classification of tokens— the token. The distinction only really becomes noticeable when preprocessing-tokens that are not also tokens occur in the source. This can occur for pp-number and the "each non-white-space character that cannot be one of the above" and is discussed elsewhere. There does not appear to be a worthwhile benefit in educating developers about preprocessing-tokens.

pp-number 927 syntax
preprocessing token 770 syntax

Summary

The following two sections provide background on those low-level visual processing operations that might be applicable to source code. The first section covers what is sometimes called *early vision*. This phase of vision is carried out without apparent effort. The possibilities for organizing the visual appearance of source code to enable it to be visually processed with no apparent effort are discussed. At this level the impact of individual characters is not considered, only the outline image formed by sequences (either vertically or horizontally) of characters. The second section covers eye movement in reading. This deals with the processing of individual, horizontal sequences of characters. To some extent the form of these sequences is under the control of the developer. Identifiers (whose spelling is under developer-control) and space characters make up a large percentage of the characters on a line.

The early vision factors that appear applicable to C source code are proximity, edge detection, and distinguishing features. The factors affecting eye movement in reading are practice related. More frequently encountered words are processed more quickly and knowledge of the frequency of letter sequences is used to decide where to move the eyes next.

770 reading practice
770 letter detection

The discussion assumes a 2-D visual representation; although 3-D visualization systems have been developed[1285] they are still in their infancy.

1 Early vision

One of the methods used by these coding guidelines to achieve their stated purpose is to make recommendations that reduce the cognitive load needed to read source code. This section provides an overview of some of the operations the human visual system can perform without requiring any apparent effort. The experimental evidence[1130] suggests that the reason these operations do not create a cognitive load is that they occur before the visual stimulus is processed by the human cognitive system. The operations occur in what is known as *early vision*. Knowledge of these operations may be of use in deciding how to organize the visible appearance of token sequences (source code layout).

vision early

0 coding guidelines introduction

The display source code uses a subset of the possible visual operations that occur in nature. It is nonmoving, two-dimensional, generally uses only two colors, items are fully visible (they are not overlayed), and edges are rarely curved. The texture of the display is provided by the 96 characters of the source character set (in many cases a limited number of additional characters are also used). Given that human visual processing is tuned to extract important information from natural scenes,[580] it is to be expected that many optimized visual processes will not be applicable to viewing source code.

214 source character set

1.1 Preattentive processing

Some inputs to the human visual system appear to *pop-out* from their surroundings. Preattentive processing, so called because it occurs before conscious attention, is automatic and apparently effortless. The examples in Figure 770.1 show some examples of features that *pop-out* at the reader.

vision preattentive

Preattentive processing is independent of the number of distractors; a search for the feature takes the same amount of time whether it occurs with one, five, ten, or more other distractors. However, the disadvantage is that it is only effective when the features being searched for are relatively rare. When a display contains many different, distinct features (the mixed category in Figure 770.1), the *pop-out* effect does not occur. The processing abilities demonstrated in Figure 770.1 are not generally applicable to C source code for a number of reasons.

221 basic source character set

- C source code is represented using a fixed set of characters. Opportunities for introducing *graphical effects* into source code are limited. The only, universally available technique for controlling the visual appearance of source code is white space.
- While there are circumstances when a developer might want to attend to a particular identifier, or declaration, in general there are no constructs that need to *pop-out* to all readers of the source. Program development environments may highlight (using different colors) searched for constructs, dependencies between constructs, or alternative representations (for instance, call graphs), but these are temporary requirements that change over short periods of time, as the developer attempts to comprehend source code.

1.2 Gestalt principles

Founded in 1912 the Gestalt school of psychology proposed what has become known as the *Gestalt laws of perception* (*gestalt* means *pattern* in German); they are also known as the *laws of perceptual organization*. The underlying idea is that the whole is different from the sum of its parts. These so-called *laws* do not have the rigour expected of a scientific law, and really ought to be called by some other term (e.g., principle). The following are some of the more commonly occurring principles

gestalt principles

- **Proximity**: Elements that are close together are perceptually grouped together (see Figure 770.2).

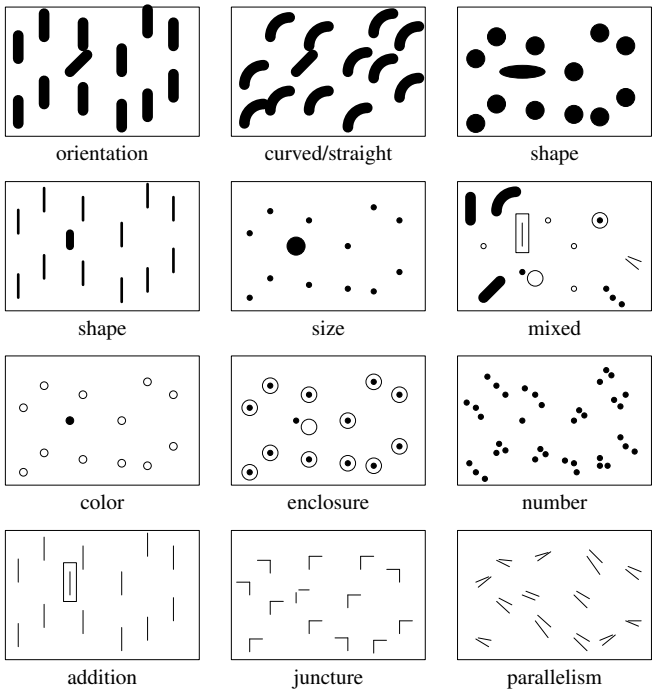


Figure 770.1: Examples of features that may be preattentively processed (parallel lines and the junction of two lines are the odd ones out). Adapted from Ware.[1441]

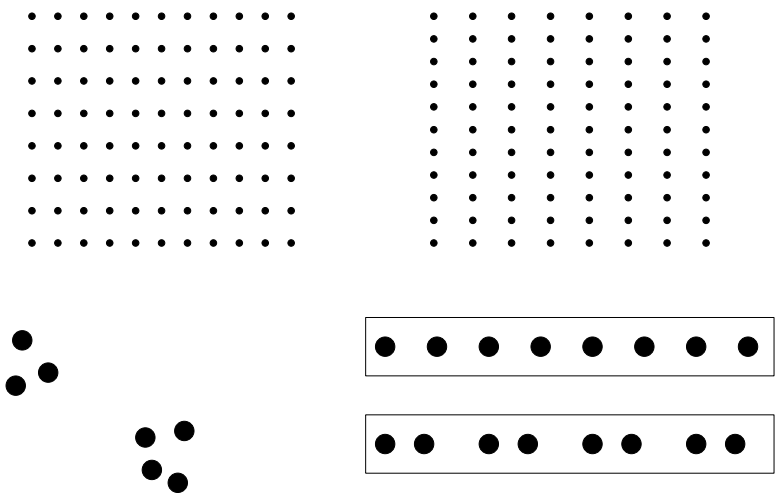


Figure 770.2: Proximity— the horizontal distance between the dots in the upper left image is less than the vertical distance, causing them to be perceptually grouped into lines (the relative distances are reversed in the upper right image).

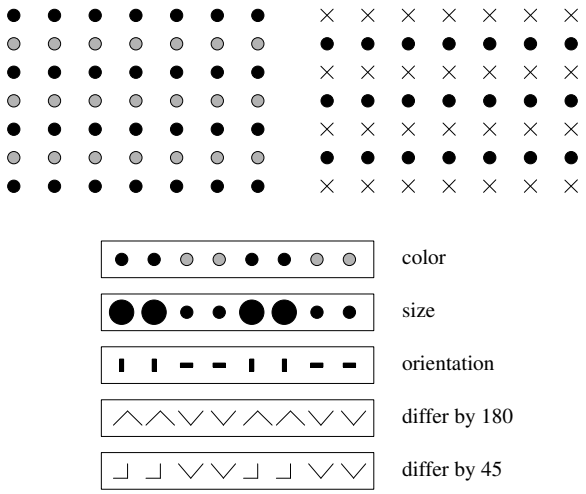


Figure 770.3: Similarity— a variety of dimensions along which visual items can differ sufficiently to cause them to be perceived as being distinct; rotating two line segments by 180° does not create as big a perceived difference as rotating them by 45°.

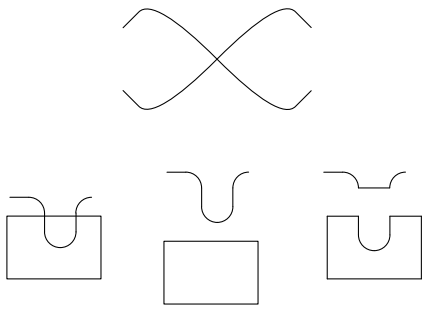


Figure 770.4: Continuity— upper image is perceived as two curved lines; the lower-left image is perceived as a curved line overlapping a rectangle rather than an angular line overlapping a rectangle having a piece missing (lower-right image).

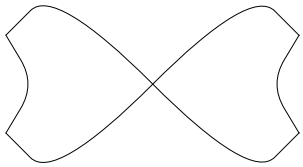


Figure 770.5: Closure— when the two perceived lines in the upper image of Figure 770.4 are joined at their end, the perception changes to one of two cone-shaped objects.

- **Similarity**: Elements that share a common attribute can be perceptually grouped together (see Figure 770.3).
- **Continuity**, also known as **Good continuation**: Lines and edges that can be seen as smooth and continuous are perceptually grouped together (see Figure 770.4). continuation gestalt principle of
- **Closure**: Elements that form a closed figure are perceptually grouped together (see Figure 770.5).
- **Symmetry**: Treating two, mirror image lines as though they form the outline of an object (see Figure 770.6). This effect can also occur for parallel lines.
- **Other** principles include grouping by connectedness, grouping by common region, and synchrony.[1047]

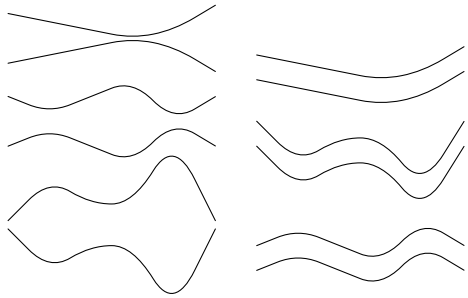


Figure 770.6: Symmetry and parallelism— where the direction taken by one line follows the same pattern of behavior as another line.

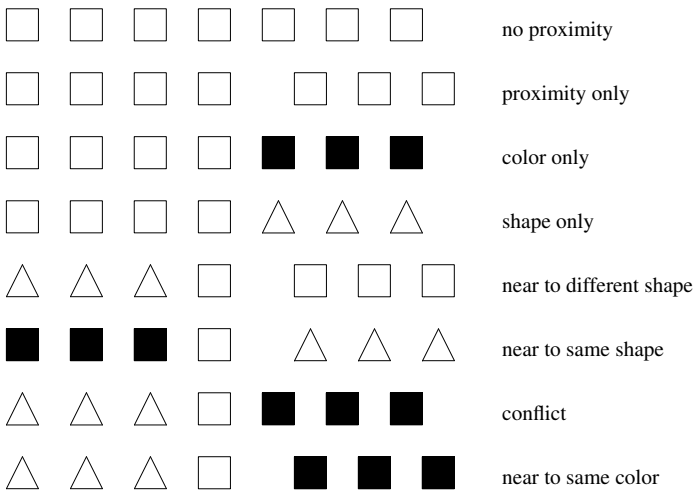


Figure 770.7: Conflict between proximity, color, and shape. Based on Quinlan.[1132]

The organization of visual grouping of elements in a display, using these principles, is a common human trait. However, when the elements in a display contain instances of more than one of these perceptual organization principles, people differ in their implicit selection of principle used. A study by Quinlan and Wilton[1132] found that 50% of subjects grouped the elements in Figure 770.7 by proximity and 50% by similarity. They proposed the following, rather incomplete, algorithm for deciding how to group elements:

1. Proximity is used to initially group elements.
2. If there is a within-group attribute mismatch, but a match between groups, people select between either a proximity or a similarity grouping (near to different shape in Figure 770.7).
3. If there is a within-group and between-group attribute mismatch, then proximity is ignored. Grouping is then often based on color rather than shape (near to same color and near to same shape in Figure 770.7).

Recent work by Kubovy and Gepshtein[779] has tried to formulate an equation for predicting the grouping of rows of dots. Will the grouping be based on proximity or similarity? They found a logarithmic relationship between dot distance and brightness that is a good predictor of which grouping will be used.

The symbols available to developers writing C source provide some degree of flexibility in the control of its visual appearance. The appearance is also affected by parameters outside of the developers' control—for instance, line and intercharacter spacing. While developers may attempt to delineate sections of source

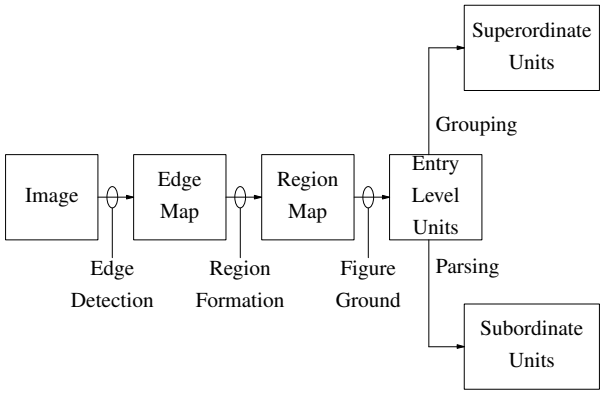


Figure 770.8: A flowchart of Palmer and Rock's[1047] theory of perceptual organization.

using white space and comments, the visual impact of the results do not usually match what is immediately apparent in the examples of the Gestalt principles given above. While instances of these principles may be used in laying out particular sequences of code, there is no obvious way of using them to create generalized layout rules. The alleged benefits of particular source layout schemes invariably depend on practice (a cost). The Gestalt principles are preprogrammed (i.e., there is no conscious cognitive cost). These coding guidelines cannot perform a cost/benefit analysis of the various code layout rules because your author knows of no studies, using experienced developers, investigating this topic.

770 reading practice

1.3 Edge detection

The striate cortex is the visual receiving area of the brain. Neurons within this area respond selectively to the orientation of edges, the direction of stimulus movement, color along several directions, and other visual stimuli. In Palmer and Rock's[1047] theory of perceptual organization, edge detection is the first operation performed on the signal that appears as input to the human visual system. After edges are detected, regions are formed, and then figure–ground principles operate to form entry-level units (see Figure 770.8).

C source is read from left to right, top to bottom. It is common practice to precede the first non-white-space character on a sequence of lines to start at the same horizontal position. This usage has been found to reduce the effort needed to visually process lines of code that share something in common; for instance, statement indentation is usually used to indicate block nesting.

Edge detection would appear to be an operation that people can perform with no apparent effort. An edge can also be used to speed up the search for an item if it occurs along an edge. In the following sequences of declarations, less effort is required to find a particular identifier in the second two blocks of declarations. In the first block the reader first has to scan a sequence of tokens to locate the identifier being declared. In the other two blocks the locations of the identifiers are readily apparent. Use of edges is only part of the analysis that needs to be carried out when deciding what layout is likely to minimize cognitive effort. These analyses are given for various constructs elsewhere.

1707 statement visual layout
1348 declaration visual layout

```
1   /* First block. */
2   int glob;
3   unsigned long a_var;
4   const signed char ch;
5   volatile int clock_val;
6   void *free_mem;
7   void *mem_free;
8
9   /* Second block. */
10  int                glob;
11  unsigned long      a_var;
12  const signed char  ch;
```

```
volatile int       clock_val;
void *             free_mem;
void               *mem_free;

/* Third block. */
              int  glob;
    unsigned long  a_var;
const signed char  ch;
     volatile int  clock_val;
           void *  free_mem;
             void  *mem_free;
```

reading kinds of 770

Searching is only one of the visual operations performed on source. Systematic line-by-line, token-by-token reading is another. The extent to which the potentially large quantities of white space introduced to create edges increases the effort required for systematic reading is unknown. For instance, the second block (previous code example) maintains the edge at the start of the lines at which systematic reading would start, but at the cost of requiring a large saccade to the identifier. The third block only requires a small saccade to the identifier, but there is no edge to aid in the location of the start of a line.

Saccade 770

1.4 Reading practice

reading practice

expertise 0

A study by Kolers and Perkins[761] offers some insight into the power of extended practice. In this study subjects were asked to read pages of text written in various ways; pages contained, normal, reversed, inverted, or mirrored text.

```
Expectations can also mislead us; the unexpected is always hard to
perceive clearly.  Sometimes we fail to recognize an object because we
saw tI  .eb ot serad eh sa yzal sa si nam yreve taht dias ecno nosremE
si tI  .ekam ot detcepxe eb thgim natiruP dnalgnE weN a ekatsim fo dnik eht
These are but a few of the reasons for believing that a
person cannot be conscious of all his mental processes.
Many other reasons can be
                                Several years ago a professor who teaches psychology at a
                          large university had to ask his assistant, a young man of great
                                                                             intelligence
```

power law of learning 0

The time taken for subjects to read a page of text in a particular orientation was measured. The more pages subjects read, the faster they became. This is an example of the power law of learning. A year later Kolers[760] measured the performance of the same subjects, as they read more pages. Performance improved with practice, but this time the subjects had past experience and their performance started out better and improved more quickly (see Figure 770.9). These results are similar to those obtained in the letter-detection task.

letter detection 770

Just as people can learn to read text written in various ways, developers can learn to read source code laid out in various ways. The important issue is not developers' performance with a source code layout they have extensive experience reading, but their performance on a layout they have little experience reading. For instance, how quickly can they achieve a reading performance comparable to that achieved with a familiar layout (based on reading and error rate). The ideal source code layout is one that can be quickly learned and has a low error rate (compared with other layouts).

reading practice 770

Unfortunately there are no studies, using experienced developers, that compare the effects of different source code layout on reading performance. Becoming an experienced developer can be said to involve learning to read source that has been laid out in a number of different ways. The visually based guidelines in this book do not attempt to achieve an optimum layout, rather they attempt to steer developers away from layouts that are likely to be have high error rates.

Many developers believe that the layout used for their own source code is optimal for reading by themselves, and others. It may be true that the layout used is optimal for the developer who uses it, but the reason for this

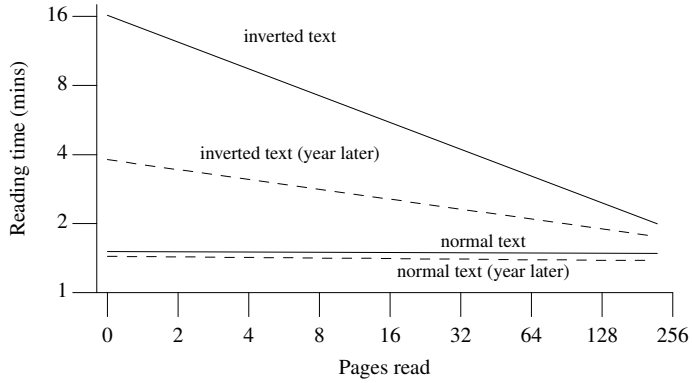


Figure 770.9: The time taken for subjects to read a page of text in a particular orientation, as they read more pages. Results are for the same six subjects in two tests more than a year apart. Based on Kolers.[760]

is likely to be practice-based rather than any intrinsic visual properties of the source layout. Other issues associated with visual code layout are discussed in more detail elsewhere.

1348 declaration visual layout
1707 statement visual layout

1.5 Distinguishing features

distinguishing features

A number of studies have found that people are more likely to notice the presence of a distinguishing feature than the absence of a distinguishing feature. This characteristic affects performance when searching for an item when it occurs among visually similar items. It can also affect reading performance— for instance, substituting an *e* for a *c* is more likely to be noticed than substituting a *c* for an *e*.

A study by Treisman and Souther[1363] found that visual searches were performed in parallel when the target included a unique feature (search time was not affected by the number of background items), and searches were serial when the target had a unique feature missing (search time was proportional to the number of background items). These results were consistent with Treisman and Gelade's[1364] feature-integration theory.

What is a unique feature? Treisman and Souther investigated this issue by having subjects search for circles that differed in the presence or absence of a gap (see Figure 770.10). The results showed that subjects were able to locate a circle containing a gap, in the presence of complete circles, in parallel. However, searching for a complete circle, in the presence of circles with gaps, was carried out serially. In this case the gap was the unique feature. Performance also depended on the proportion of the circle taken up by the gap.

As discussed in previous subsections, C source code is made up of a fixed number of different characters. This restricts the opportunities for organizing source to take advantage of the search asymmetry of preattentive processing. It is important to remember the preattentive nature of parallel searching; for instance, comments are sometimes used to signal the presence of some construct. Reading the contents of these comments would require attention. It is only their visual presence that can be a distinguishing feature from the point of view of preattentive processing. The same consideration applies to any organizational layout using space characters. It is the visual appearance, not the semantic content that is important.

1.6 Visual capacity limits

A number of studies have looked at the capacity limits of visual processing.[426, 1419] Source code is visually static, that is it does not move under the influence of external factors (such as the output of a dynamic trace of an executing program might). These coding guidelines make the assumption that the developer-capacity bottleneck occurs at the semantic level, not the visual processing stage.

2 Reading (eye movement)

Reading eye movement

While C source code is defined in terms of a sequence of ordered lines containing an ordered sequence of characters, it is rarely read that way by developers. There is no generally accepted theory for how developers read source code, at the token level, so the following discussion is necessarily broad and lacking in detail. Are

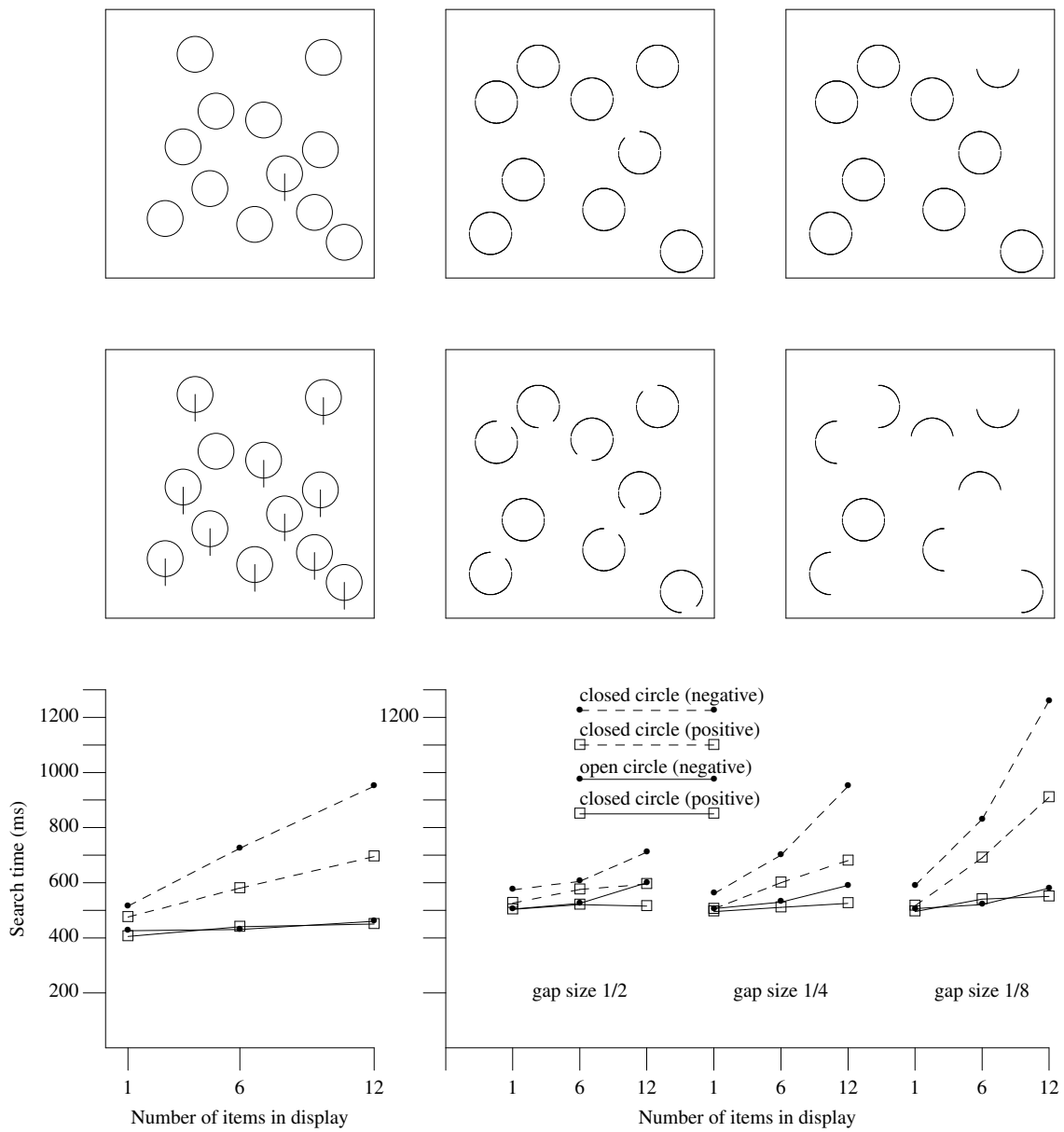


Figure 770.10: Examples of unique items among visually similar items. Those at the top include an item that has a distinguishing feature (a vertical line or a gap); those underneath them include an item that is missing this distinguishing feature. Graphs represent time taken to locate unique items (positive if it is present, negative when it is not present) when placed among different numbers of visibly similar distractors. Based on displays used in the study by Treisman and Sother.[1363]

there any organizational principles of developers' visual input that can be also be used as visual organizational principles for C source code?

Developers talk of reading source code; however, reading C source code differs from reading human language prose in many significant ways, including:

- It is possible, even necessary, to create new words (identifiers). The properties associated with these words are decided on by the author of the code. These words might only be used within small regions of text (their scope); their meaning (type) and spelling are also under the control of the original developer.
- Although C syntax specifies a left-to-right reading order (which is invariably presented in lines that read from the top, down), developers sometimes find it easier to comprehend statements using either a right-to-left reading, or even by starting at some subcomponent and working out (to the left and right) or lines reading from the bottom, up.
- Source code is not designed to be a spoken language; it is rare for more than short snippets to be verbalized. Without listeners, developers have not needed to learn to live (write code) within the constraints imposed by realtime communication between capacity-limited parties.
- The C syntax is not locally ambiguous. It is always possible to deduce the syntactic context, in C, using a lookahead of a single word[770.1] (the visible source may be ambiguous through the use of the preprocessor, but such usage is rare and strongly recommended against). This statement is not true in C++ where it is possible to write source that requires looking ahead an indefinite number of words to disambiguate a localized context.
- In any context a word has a single meaning. For instance, it is not necessary to know the meaning (after preprocessing) of `a`, `b` and `c`, to comprehend `a=b+c`. This statement is not true in computer languages that support overloading, for instance C++ and Java.
- Source code denotes operations on an abstract machine. Individually the operations have no external meaning, but sequences of these operations can be interpreted as having an equivalence to a model of some external *real-world* construct. For instance, the expression `a=b+c` specifies the abstract machine operations of adding `b` to `c` and storing the resulting value in `a`; its interpretation (as part of a larger sequence of operations) might be *move on to the next line of output*. It is this semantic mapping that creates cognitive load while reading source code. When reading prose the cognitive load is created by the need to disambiguate word meaning and deduce a parse using known (English or otherwise) syntax.

Reading and writing is a human invention, which until recently few people could perform. Consequently, human visual processing has not faced evolutionary pressure to be good at reading.

While there are many differences between reading code and prose, almost no research has been done on reading code and a great deal of research has been done on reading prose. The models of reading that have been built, based on the results of prose-related research, provide a starting point for creating a list of issues that need to be considered in building an accurate model of code reading. The following discussion is based on papers by Rayner[1147] and Reichle, Rayner, and Pollatsek.[1154]

Saccade

During reading, a person's eyes make short rapid movements. These movements are called *saccades* and take 20 ms to 50 ms to complete. No visual information is extracted during these saccades and readers are not consciously aware that they occur. A saccade typically moves the eyes forward 6 to 9 characters. Between saccades the eyes are stationary, typically for 200 ms to 250 ms (a study of consumer eye movements[1086] while comparing multiple brands found a fixation duration of 354 ms when subjects were under high time pressure and 431 ms when under low time pressure). These stationary time periods are called *fixations*. Reading can be compared to watching a film. In both cases a stationary image is available for information

[770.1] There is one exception—for the token sequence `void func (a, b, c, d, e, f, g)`. It is not known whether `func` is a declaration of a prototype or a function definition until the token after the closing parenthesis is seen.

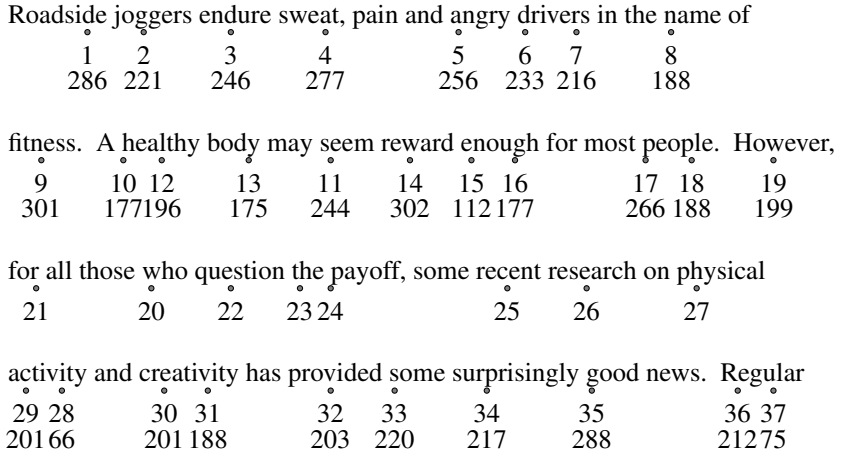


Figure 770.11: A passage of text with eye fixation position (dot under word), fixation sequence number, and fixation duration (in milliseconds) included. Adapted from Reichle, Pollatsek, Fisher, and Rayner[1153] (timings on the third line are missing in the original).

extraction before it is replaced by another (4–5 times a second in one case, 50–60 times a second in the other). However, in the case of a film the updating of individual images is handled by the projector while the film's director decides what to look at next; but during reading a person needs to decide what to look at next and move the eyes to that location.

The same reader can show considerable variation in performing these actions. Saccades might move the eyes by one character, or 15 to 20 characters (the duration of a saccade is influenced by the distance covered, measured in degrees). Fixations can be shorter than 100 ms or longer than 400 ms (they can also vary between languages[1036]). The content of the fixated text has a strong effect on reader performance.

The eyes do not always move forward during reading— 10% to 15% of saccades move the eyes back to previous parts of the text. These backward movements, called *regressions*, are caused by problems with linguistic processing (e.g., incorrect syntactic analysis of a sentence) and oculomotor error (for instance, the eyes overshot their intended target).

garden path sentence 770

visual field

Saccades are necessary because the eyes' field of view is limited. Light entering the eyes falls on the retina, where it hits light-sensitive cells. These cells are not uniformly distributed, but are more densely packed in the center of the retina. This distribution of light sensitive cells divides the visual field (on the retina) into three regions: foveal (the central 2°s, measured from the front of the eye looking toward the retina), parafoveal (extending out to 5°s), and peripheral (everything else). Letters become increasingly difficult to identify as their angular distance from the center of the fovea increases.

A reader has to perform two processes during the fixation period: (1) identify the word (or sequence of letters forming a partial word) whose image falls within the foveal and (2) plan the next saccade (when to make it and where to move the eyes). Reading performance is speed limited by the need to plan and perform saccades. If the need to saccade is removed by presenting words at the same place on a display, there is a threefold speed increase in reading aloud and a twofold speed increase in silent reading. The time needed to plan and perform a saccade is approximately 180 ms to 200 ms (known as the *saccade latency*), which means that the decision to make a saccade occurs within the first 100 ms of a fixation. How does a reader make a good saccade decision in such a short period of time?

The contents of the parafoveal region are partially processed during reading. The parafoveal region increases a reader's perceptual span. When reading words written using alphabetic characters (e.g., English or German), the perceptual span extends from 3 to 4 characters on the left of fixation to 14 to 15 letters to the right of fixation. This asymmetry in the perceptual span is a result of the direction of reading, attending to letters likely to occur next being of greater value. Readers of Hebrew (which is read right-to-left) have

a perceptual span that has opposite asymmetry (in bilingual Hebrew/English readers the direction of the asymmetry depends on the language being read, showing the importance of attention during reading[1100]).

The process of reading has attracted a large number of studies. The following general points have been found to hold:

- The perceptual span does not extend below the line being read. Readers' attention is focused on the line currently being read.
- The size of the perceptual span is fairly constant for similar alphabetic orthographies (graphical representation of letters).
- The characteristics of the writing system affect the asymmetry of the perceptual span and its width. For instance, the span can be smaller for Hebrew than English (Hebrew words can be written without the vowels, requiring greater effort to decode and plan the next saccade). It is also much smaller for writing systems that use ideographs, such as Japanese (approximately 6 characters to the right) and Chinese. 792 orthography
- The perceptual span is not hardwired, but is attention-based. The span can become smaller when the fixated words are difficult to process. Also readers obtain more information in the direction of reading when the upcoming word is highly predictable (based on the preceding text).
- Orthographic and phonological processing of a word can begin prior to the word being fixated. 792 orthography 792 phonology
- Words that can be identified in the parafovea do not have to be fixated and can be skipped. Predictable words are skipped more than unpredictable words, and short function words (like *the*) are skipped more than content words.

The processes that control eye movement have to decide where (to fixate next) and when (to move the eyes). These processes sometimes overlap and are made somewhat independently (see Figure 770.11).

Where to fixate next. Decisions about where to fixate next seem to be determined largely by low-level visual cues in the text, as follows.

- Saccade length is influenced by the length of both the fixated word and the word to the right of fixation.
- When readers do not have information about where the spaces are between upcoming words, saccade length decreases and reading rate slows considerably.
- Although there is some variability in where the eyes land on a word, readers tend to make their first fixation about halfway between the beginning and the middle of a word.
- While contextual constraints influence skipping (highly predictable words are skipped more than unpredictable words), contextual constraints have little influence on where the eyes land in a word (however, recent research[814] has found some semantic-context effects influence eye landing sites).
- The landing position on a word is strongly affected by the launch site (the previous landing position). As the launch site moves further from the target word, the distribution of landing positions shifts to the left and becomes more variable.

When to move the eyes. The ease or difficulty associated with processing a word influences when the eyes move, as follows.

- There is a spillover effect associated with fixating a low-frequency word; fixation time on the next word increases.
- Although the duration of the first fixation on a word is influenced by the frequency of that word, the duration of the previous fixation (which was not on that word) is not.

- High-frequency words are skipped more than low-frequency words, particularly when they are short and the reader has fixated close to the beginning of the word.
- Highly predictable (based on the preceding context) words are fixated for less time than words that are not so predictable. The strongest effects of predictability on fixation time are not usually as large as the strongest frequency effects. Word predictability also has a strong effect on word skipping.

2.1 Models of reading

It is generally believed that eye movements follow visual attention. This section discusses some of the models of eye movements that have been proposed and provides some of the background theory needed to answer questions concerning optimal layout of source code. An accurate, general-purpose model of eye movement would enable the performance of various code layout strategies to be tested. Unfortunately, no such model is available. This book uses features from three models, which look as if they may have some applicability to how developers read source. For a comparison of the different models, see Reichle, Rayner and Pollatsek.[1154]

2.1.1 Mr. Chips

Mr. Chips

Mr. Chips[830] is an ideal-observer model of reading (it is also the name of a computer program implemented in C) which attempts to calculate the distance, measured in characters, of the next saccade. (It does not attempt to answer the question of when the saccade will occur.) It is an idealized model of reading in that it optimally combines three sources of information (it also includes a noise component, representing imperfections in oculomotor control):

1. Visual data obtained by sampling the text through a *retina*, which has three regions mimicking the behavior of those in the human eye.
2. Lexical knowledge, consisting of a list of words and their relative frequencies (English is used in the published study).
3. Motor knowledge, consisting of statistical information on the accuracy of the saccades made.

Mr. Chips uses a single optimization principle— entropy minimization. All available information is used to select a saccade distance that minimizes the uncertainty about the current word in the visual field (ties are broken by picking the largest distance). Executing the Mr. Chips program shows it performing regressive saccades, word skips, and selecting viewing positions in words, similar to human performance.

Mr. Chips is not intended to be a model of how humans read, but to establish the pattern of performance when available information is used optimally. It is not proposed that readers perform entropy calculations when planning saccades. There are simpler algorithms using a small set of heuristics that perform close to the entropy minimization ideal (see Figure 770.12).

The eyes' handling of visual data and the accuracy of their movement control are physical characteristics. The lexical knowledge is a characteristic of the environment in which the reader grew up. A person has little control over the natural language words they hear and see, and how often they occur. Source code declarations create new words that can then occur in subsequent parts of the source being worked on by an individual. The characteristics of these words will be added to developers' existing word knowledge. Whether particular, code-specific letter sequences will be encountered sufficiently often to have any measurable impact on the lexicon a person has built up over many years is not known (see Figure 792.16).

2.1.2 The E-Z Reader model

The E-Z Reader model of eye movement control in reading is described by Reichle, Pollatsek, Fisher, and Rayner.[1153] It aims to give an account of how cognitive and lexical processes influence the eye movements of skilled readers. Within this framework it is the most comprehensive model of reading available. An important issue ignored by this model is higher order processing. (The following section describes a model that attempts to address cognitive issues.) For instance, in the sentence "Since Jay always jogs a mile

garden path sentence

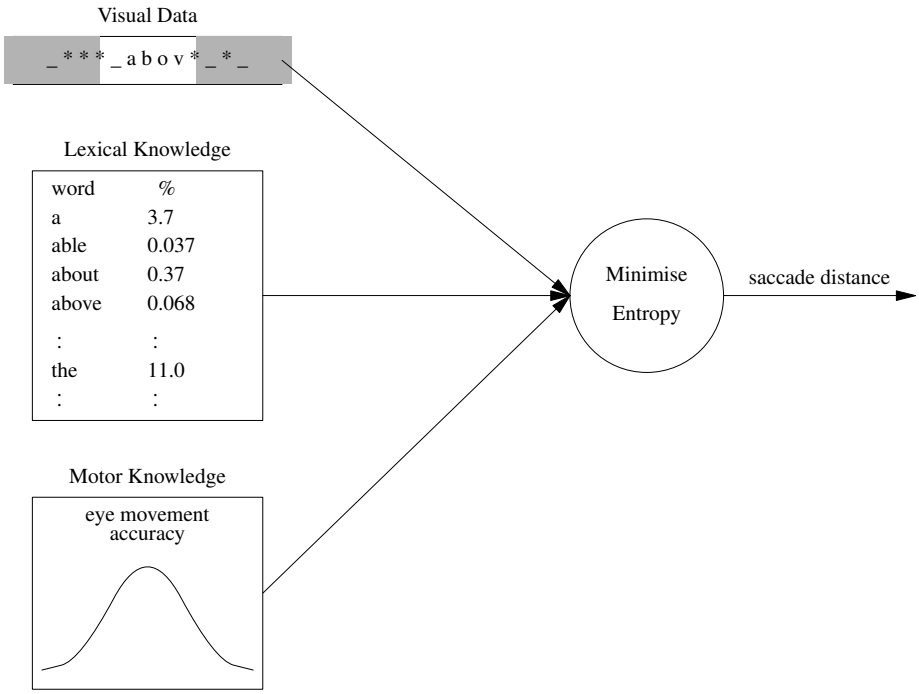


Figure 770.12: Mr. Chips schematic. The shaded region in the visual data is the parafoveal; in this region individual letters (indicated by stars) can only be distinguished from spaces (indicated by underscores). Based on Legge et al.[830]

seems like a short distance." readers experience a disruption that is unrelated to the form or meaning of the individual words. The reader has been led down a syntactic *garden path*; initially parsing the sentence so that *a mile* is the object of *jogs* before realizing that *a mile* is the subject of *seems*. Also it does not attempt to model the precise location of fixations.

The aspect of this model that is applicable to reading source code is the performance dependency, of various components to the frequency of the word being processed (refer to Figure 770.11). The *familiarity check* is a quick assessment or whether word identification is imminent, while *completion of lexical access* corresponds to a later stage when a word's identity has been determined.

2.1.3 EMMA

EMMA [1193] is a domain-independent model that relates higher-level cognitive processes and attention shifts with lower-level eye movement behavior. EMMA is based on many of the ideas in the E-Z model and uses ACT-R[35] to model cognitive processes. EMMA is not specific to reading and has been applied to equation-solving and visual search. EMMA

The *spotlight* metaphor of visual attention, used by EMMA, selects a single region of the visual field for processing. Shifting attention to a new visual object requires that it be encoded into an internal representation. The time, T_{enc}, needed to perform this encoding is:

$$T_{enc} = K(-\log f_i)e^{k\theta_i} \tag{770.1}$$

where f_i represents the frequency of the visual object being encoded (a value between 0.0 and 1.0), θ_i is its visual angle from the center of the current eye position, and K and k are constants.

The important components of this model, for these coding guidelines, are the logarithmic dependency on word frequency and the exponential decay based on the angle subtended by the word from the center of vision.

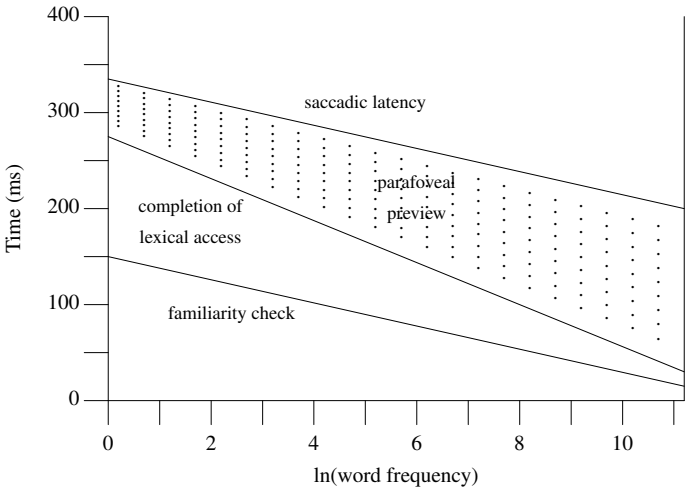


Figure 770.13: How preview benefit is affected by word frequency. The bottom line denotes the time needed to complete the familiarity check, the middle line the completion of lexical access, and the top line when the execution of the eye movement triggered by the familiarity check occurs. Based on Reichle, Pollatsek, Fisher, and Rayner.[1153]

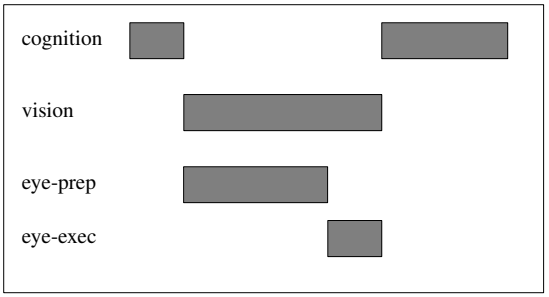


Figure 770.14: Example case of EMMA's control flow. Adapted from Salvucci.[1193]

2.2 Individual word reading (English, French, and more?)

word reading individual

When presented with a single word, or the first word in a sentence, studies have found that readers tend to pick an eye fixation point near the center of that word. The so-called *preferred viewing location* is often toward the left of center. It had been assumed that selecting such a position enabled the reader to maximize information about the letters in the word (fixating near the center would enable a reader to get the most information out of their eye's available field of view). Clark and O'Regan[245] performed a statistical analysis of several English word corpuses and a French word corpus. They assumed the reader would know the first and last letters of a word, and would have a choice of looking anywhere within a word to obtain information on more letters. (It was assumed that reliable information on two more letters would be obtained.)

Knowing only a few of the letters of a word can create ambiguity because there is more than one, human language, word containing those letters at a given position. For instance, some of the words matched by *s*at****d* include *scattered*, *spattered*, and *stationed*. The results (see Figure 770.15) show that word ambiguity is minimized by selecting two letters near the middle of the word. Clark and O'Regan do not give any explanation for why English and French words should have this property, but they do suggest that experienced readers of these two languages make use of this information in selecting the optimal viewing position within a word.

word frequency 792

There are a number of experimental results that cannot be explained by an eye viewing position theory based only on word ambiguity minimization. For instance, the word frequency effect shows that high-

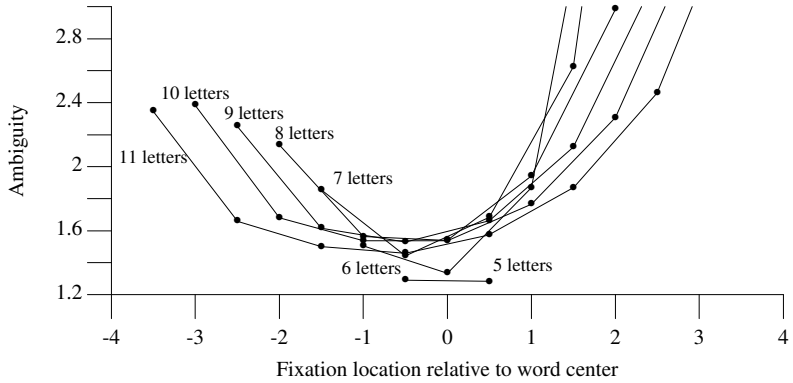


Figure 770.15: The ambiguity of patterns defined by the first and last letter and an interior letter pair, as a function of the position of the first letter of the pair. Plots are for different word lengths using the 65,000 words from CLAWS[822] (as used by the `aspell` tool). The fixation position is taken to be midway between the interior letter pair.

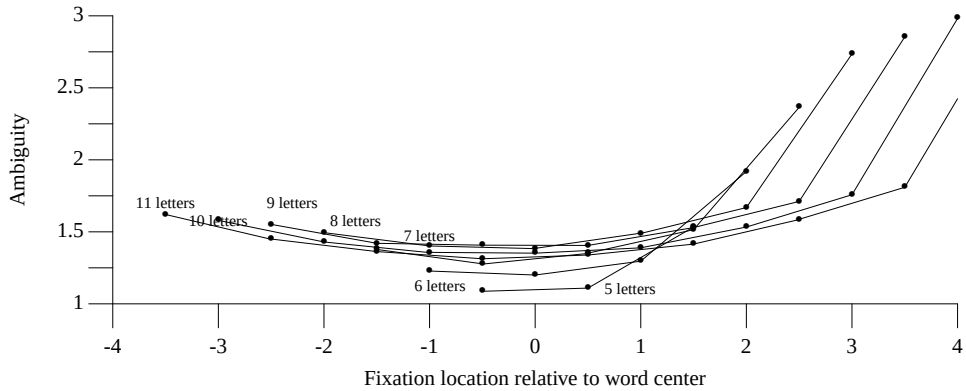


Figure 770.16: The ambiguity of source code identifiers, which can include digits as well as alphabetic characters. Plots are for different identifier lengths. A total of 344,000 identifiers from the visible form of the `.c` files were used.

frequency words are more easily recognized than low-frequency words. The ambiguity data shows the opposite effect. While there must be other reading processes at work, Clark and O'Regan propose that ambiguity minimization is a strong contributor to the optimal viewing position.

The need to read individual identifiers in source code occurs in a number of situations. Developers may scan down a list of identifiers looking for (1) the declaration of a particular identifier (where it is likely to be the last sequence of letters on a line) or (2) a modification of a particular identifier (where it is likely to be the first non-space character on a line).

If developers have learned that looking at the middle of a word maximizes their information gain when reading English text, it is likely this behavior will be transferred to reading source code. Identifiers in source code are rarely existing, human language, words. The extent to which experienced developers learn to modify their eye movements (if any modification is necessary) when reading source code is unknown. If we assume there is no significant change in eye movement behavior on encountering identifiers in source code, it can be used to predict the immediate information available to a developer on first seeing an identifier. Knowing this information makes it possible to select identifier spellings to minimize ambiguity with respect to other identifiers declared in the same program. This issue is discussed elsewhere.

792 identifier syntax

Calculating the ambiguity for different positions within C source code identifiers shows (see Figure 770.16) that the ambiguity is minimized near the center of the identifier and rises rapidly toward the end. However, there is a much smaller increase in ambiguity, compared to English words, moving toward the beginning

of the identifier. Why English speakers and developers (the source code used for these measurements is likely to be predominantly written by English speakers but not necessarily native speakers) should create words/identifiers with this ambiguity minimization property is not known.

If native-English speakers do use four letters worth of information to guide identifier lookup, will they be misled by their knowledge of English words? Of the 344,000 unique identifiers (41.6% contained between 5 and 11 characters) in the `.c` files, only 0.45% corresponded to words in the CLAWS list of 65,000 words. The letter pattern counts showed the words containing a total of 303,518 patterns, to which the list of identifiers added an additional 1,576,532 patterns. The identifiers contained letters that matched against 166,574 word patterns (9.5% for center pair) and matched against 608,471 patterns that were unique to identifiers (8.1% for center pair).

These results show that more than 80% of letter patterns appearing in identifiers do not appear in English words. Also, identifier letter patterns are three times more likely to match against a pattern that is unique to identifiers than a pattern that occurs in an English word. In most cases developers will not be misled into thinking of an English word because four-letter patterns in identifiers do not frequently occur in English words.

2.3 White space between words

words white space between
reading 770 kinds of

The use of white space between tokens in source code is a controversial subject. The use of white space is said to affect *readability*, however that might be measured. The different reasons a developer has for reading source code, and the likely strategies adopted are discussed elsewhere.

Is the cost of ownership of source code that contains a space character, where permitted, between every identifier and operator/punctuator[770.2] less than or greater than the cost of ownership of source code that does not contain such space characters? This subsection discusses the issues; however, it fails to reach a definitive conclusion.

Readers of English take it for granted that a space appears between every word in a line of text. This was not always the case. Spaces between words started to be used during the time of Charlemagne (742–814); however, as late as the seventeenth century there was still some irregularity in how spaces were used to separate letters.[138] The spread of Latin to those less familiar with it, and the availability of books (through the introduction of the printing press) to people less skilled in reading, created a user-interface problem. Spaces between words simplified life for occasional users of Latin and improved the user friendliness of books for intermittent readers. The written form of some languages do not insert spaces between words (e.g., Japanese and Thai), while other written forms add some spaces but also merge sequences of words to form a single long word (e.g., German and Dutch). Algorithms for automating the process of separating words in unspaced text is an active research topic.[1045]

ReadersofEnglishdonotneedspacesbetweenwords.Isitsimplylackofpracticethatreducesreadingrate?Thestudybyby Kolers[761] showed what could be achieved with practice. Readers of source code do not need spaces either (in a fewcontexts the syntax requires them) `a=b+c`. The difference between English prose and source code is that identifier words are always separated by other words (operators and punctuation) represented by characters that cannot occur in an identifier.

A study by Epelboim, Booth, Ashkenazy, Taleghani, and Steinmans[392] did not just simply remove the spaces from between words, they also added a variety of different characters between the words (shaded boxes, digits, lowercase Greek letters, or lowercase Latin letters). Subjects were not given any significant training on reading the different kinds of material.

Epelboim[392]

The following filler-placements were used (examples with digit fillers are shown in parentheses):

1. *Normal: Normal text (this is an example);*
2. *Begin: A filler at the beginning of each word, spaces preserved (1this 3is 7an 2example);*

[770.2] In some cases a space character is required between tokens; for instance, the character sequence `const int i` would be treated as a single identifier if the space characters were not included.

3. *End: A filler after the end of each word, spaces preserved (this1 is3 an7 example2);*
4. *Surround: Fillers surrounding each word, spaces preserved (9this1 4is3 6an7 8example2);*
5. *Fill-1: A filler filling each space (9this2is5an8example4);*
6. *Fill-2: Two fillers filling each space (42this54is89an72example39);*
7. *Unspaced: Spaces removed, no fillers (thisisanexample).*

Table 770.1: Mean percentage differences, compared to normal, in reading times (silent or aloud); the values in parenthesis are the range of differences. Adapted from Epelboim.[392]

Filler type	Surround	Fill-1	Fill-2	Unspaced
Shaded boxes (aloud)	4 (1–12)	—	3 (-2–9)	44 (25–60)
Digits (aloud)	26 (15–40)	26 (10–42)	—	42 (19–64)
Digits (silent)	40 (32–55)	41 (32–58)	—	52 (45–63)
Greek letters (aloud)	33 (20–47)	36 (23–45)	46 (33–57)	44 (32–53)
Latin letters (aloud)	55 (44–70)	—	74 (58–84)	43 (13–58)
Latin letters (silent)	66 (51–75)	75 (68–81)	—	45 (33–60)

Epelboim et al. interpreted their results as showing that fillers slowed reading because they interfered with the recognition of words, not because they obscured word-length information (some models of reading propose that word length is used by low-level visual processing to calculate the distance of the next saccade). They concluded that word-length information obtained by a low-level visual process that detects spaces, if used at all, was only one of many sources of information used to calculate efficient reading saccade distances.

Digits are sometimes used in source code identifiers as part of the identifier. These results suggest that digits appearing within an identifier could disrupt the reading of its complete name (assuming that the digits separated two familiar letter sequences). The performance difference when Greek letters were used as separators was not as bad as for Latin letters, but worse than digits. The reason might relate to the relative unfamiliarity of Greek letters, or their greater visual similarity to Latin letters (the letters α , δ, θ, μ, π, σ, τ, and ϕ were used). The following are several problems with applying the results of this study to reading source code.

- Although subjects were tested for their comprehension of the contents of the text (the results of anybody scoring less than 75% were excluded, the mean for those included was 89.4%), they were not tested for correctly reading the filler characters. In the case of source code the operators and punctuators between words contribute to the semantics of what is being read; usually it is necessary to pay attention to them.
- Many of the character sequences (a single character for those most commonly seen) used to represent C operators and punctuators are rarely seen in prose. There contribution to the entropy measure used to calculate saccade distances is unknown. For experienced developers the more commonly seen character sequences, which are also short, may eventually start to exhibit high-frequency word characteristics (i.e., being skipped if they appear in the parafoveal).
- Subjects were untrained. To what extent would training bring their performance up to a level comparable to the unfilled case?

A study by Kohsom and Gobet[759] used native Thai speakers, who also spoke English, as subjects (they all had an undergraduate degree and were currently studying at the University of Pittsburgh). The written form of Thai does not insert spaces between words, although it does use them to delimit sentences. In the study the time taken to read a paragraph, and the number of errors made was measured. The paragraph was in Thai or English with and without spaces (both cases) between words. The results showed no significant performance

differences between reading spaced or unspaced Thai, but there was a large performance difference between reading spaced and unspaced English.

This study leaves open the possibility that subjects were displaying a learned performance. While the Thai subjects were obviously experienced readers of unspaced text in their own language, they were not experienced readers of Thai containing spaces. The Thai subjects will have had significantly more experience reading English text containing spaces than not containing spaces. The performance of subjects was not as good for spaced English, their second language, as it was for Thai. Whether this difference was caused by subjects' different levels of practice in reading these languages, or factors specific to the language is not known. The results showed that adding spaces when subjects had learned to read without them did not have any effect on performance. Removing spaces when subjects had learned to read with them had a significant effect on performance.

In the case of expressions in source code, measurements show (see Table 770.2) that 47.7% of expressions containing two binary operators do not have any space between binary operators and their operands, while 43% of such expressions have at least one space between the binary operators and their adjacent operands.

Further studies are needed before it is possible to answer the following questions:

- Would inserting a space between identifiers and adjacent operators/punctuators reduce the source reading error rate? For instance, in `a=b*c` the `*` operator could be mistaken for the `+` operator (the higher-frequency case) or `&` operator (the lower frequency case).
- Would inserting a space between identifiers and adjacent operators/punctuators reduce the source reading rate? For instance, in `d=e[f]` the proximity of the `[` operator to the word `e` might provide immediate semantic information (the word denotes an array) without the need for another saccade.
- What impact does adding characters to a source line have on the average source reading rate and corresponding error rate (caused by the consequential need to add line breaks in some places)?
- Are the glyphs used for some characters sufficiently distinctive that inserting space characters around them has a measurable impact?

glyph 58

- Do some characters occur sufficiently frequently that experienced developers can distinguish them with the same facility in spaced and unspaced contexts?

letter detection 770

The results of the prose-reading studies discussed here would suggest that high-performance is achieved through training, not the use of spaces between words. Given that developers are likely to spend a significant amount of time being trained on (reading) existing source code, the spacing characteristics of this source would appear to be the guide to follow.

Table 770.2: Number of expressions containing two binary operators (excluding any assignment operator, comma operator, function call operator, array access or member selection operators) having the specified spacing (i.e., no spacing, *no-space*, or one or more whitespace characters (excluding newline), *space*) between a binary operator and both of its operands. *High-Low* are expressions where the first operator of the pair has the higher precedence, *Same* are expressions where the both operators of the pair have the same precedence, *Low-High* are expressions where the first operator of the pair has the lower precedence. For instance, `x + y*z` is *space no-space* because there are one or more *space* characters either side of the addition operator and *no-space* either side of the multiplication operator, the precedence order is *Low-High*. Based on the visible form of the `.c` files.

	Total	High-Low	Same	Low-High
no-space	34,866	2,923	29,579	2,364
space no-space	4,132	90	393	3,649
space space	31,375	11,480	11,162	8,733
no-space space	2,659	2,136	405	118
total	73,032	16,629	41,539	14,864

2.3.1 Relative spacing

operator relative spacing

The spacing between a sequence of tokens can be more complicated that presence/absence, it can also be relative (i.e., more spacing between some tokens than others. One consequence of relative spacing is that the eye can be drawn to preferentially associate two tokens (e.g., nearest neighbors) over other associations involving more distant tokens (see Figure 770.2).

A study by Landy and Goldstone[806] asked subjects to compute the value of expressions that contained an addition and multiplication operator (e.g., `2 + 3*4`). The spacing between the operators and adjacent operands was varied (e.g., sometimes there was more spacing adjacent to the multiplication operator than the addition, such as `5+2 * 3`).

The results showed that a much higher percentage of answers was correct when there was less spacing around the multiplication operator than the addition operator (i.e., the operands had a greater visual proximity to the multiplication operator). In this case subjects also gave the correct answer more quickly (2.6 vs. 2.9 seconds).

Relative spacing is sometimes used within source code expressions to highlight the relative precedence of binary operators. Table 770.2 shows that when relative spacing was used it occurred in a form that gave the operator with higher precedence greater proximity to its operands (compared to the operator of lower precedence).

943 operator precedence

2.4 Other visual and reading issues

There are several issues of importance to reading source code that are not covered here. Some are covered elsewhere; for instance, visual grouping by spatial location and visual recognition of identifiers. The question of whether developers should work from paper or a screen crops up from time to time. This topic is outside of the scope of these coding guidelines (see Dillon[358] for a review).

1707 grouping spatial location
792 word visual recognition

Choice of display font is something that many developers are completely oblivious to. The use of Roman, rather than Helvetica (or serif vs. sans serif), is often claimed to increase reading speed and comprehension. A study by Lange, Esterhuizen, and Beatty[332] showed that young school children (having little experience with either font) did not exhibit performance differences when either of these fonts was used. This study showed there were no intrinsic advantages to the use of either font. Whether people experience preferential exposure to one kind of font, which leads to a performance improvement through a practice effect, is not known. The issues involved in selecting fonts are covered very well in a report detailing *Font Requirements for Next Generation Air Traffic Management Systems*.[161] For a discussion of how font characteristics affect readers of different ages, see Connolly.[266]

font

A study by Pelli, Burns, Farell, and Moore[1070] showed that 2,000 to 4,000 trials were all that was needed for novice readers to reach the same level of efficiency as fluent readers in the letter-detection task. They tested subjects aged 3 to 68 with a range of different (and invented) alphabets (including Hebrew, Devanagari, Arabic, and English). Even fifty years of reading experience, over a billion letters, did not improve the efficiency of letter detection. The measure of efficiency used was human performance compared to an ideal observer. They also found this measure of efficiency was inversely proportional to letter perimetric complexity (defined as, inside and outside perimeter squared, divided by *ink* area).

letter detection

A number of source code editors highlight (often by using different colors) certain character sequences (e.g., keywords). The intended purpose of this highlighting is to improve program readability. Some source formatting tools go a stage further and highlight complete constructs (e.g., comments or function headers). A study by Gellenbeck[478] suggested that while such highlighting may increase the prominence of the construct to which it applies; it does so at the expense of other constructs.

A book by Baecker and Marcus[77] is frequently quoted in studies of source code layout. Their aim was to base the layout used on the principles of good typography (the program source code as book metaphor is used). While they proposed some innovative source visualization ideas, they seem to have been a hostage to some arbitrary typography design decisions in places. For instance, the relative frequent change of font, and the large amount of white space between identifiers and their type declaration, requires deliberate effort to align identifiers with their corresponding type declaration. While the final printed results look superficially

attractive to a casual reader, they do not appear, at least to your author, to offer any advantages to developers who regularly work with source code.

3 Kinds of reading

reading kinds of

The way in which source code is read will be influenced by the reasons for reading it. A reader has to balance goals (e.g., obtaining accurate information) with the available resources (e.g., time, cognitive resources such as prior experience, and support tools such as editor search commands).

Foraging theory[1289] attempts to explain how the behavioral adaptations of an organism (i.e., its lifestyle) are affected by the environment in which it has to perform and the constraints under which it has to operate. Pirolli and Card[1091] applied this theory to deduce the possible set of strategies people might apply when searching for information. The underlying assumption of this theory is that: faced with information-foraging tasks and the opportunity to learn and practice, cognitive strategies will evolve to maximize information gain per unit cost.

Almost no research has been done on the different information-gathering strategies (e.g., reading techniques) of software developers. These coding guidelines assume that developers will adopt many of the strategies they use for reading prose text. A review by O'Hara[1029] listed four different prose reading techniques:

O'Hara[1029]

> ***Receptive Reading***. *With this type of reading the reader receives a continuous piece of text in a manner which can be considered as approximating listening behavior. Comprehension of the text requires some portion of the already read text to be held in working memory to allow integration of meaning with the currently being read text.*
>
> ***Reflective Reading***. *This type of reading involves interruptions by moments of reflective thought about the contents of the text.*
>
> ***Skim Reading***. *This is a rapid reading style which can be used for establishing a rough idea of the text. This is useful in instances where the reader needs to decide whether the text will be useful to read or to decide which parts to read.*
>
> ***Scanning***. *This is related to skimming but refers more specifically to searching the text to see whether a particular piece of information is present or to locate a piece of information known to be in the text.*

Deimel and Naveda[341] provide a teachers' guide to program reading. The topic of visual search for identifiers is discussed in more detail elsewhere.

identifier 792
visual search

Readers do not always match up pairs of **if/else** tokens by tracing through the visible source. The source code indentation is often used to perform the matching, readers assuming that **if/else** tokens at the same indentation level are a matching pair. Incorrectly indented source code can lead to readers making mistakes.

```
1  void f(int i)
2  {
3  if (i > 8)
4     if (i < 20)
5        i++;
6  else
7     i--;
8  }
```

Example

```
1  #define mkstr(x) #x
2
3  char *p = mkstr(@); /* Implementation supports the @ extended character. */
```

For those wanting to teach code reading skills, Deimel and Naveda[341] offers an instructors guide (the examples use Ada). Studying those cases where the requirement is to minimize readability[256] can also be useful.

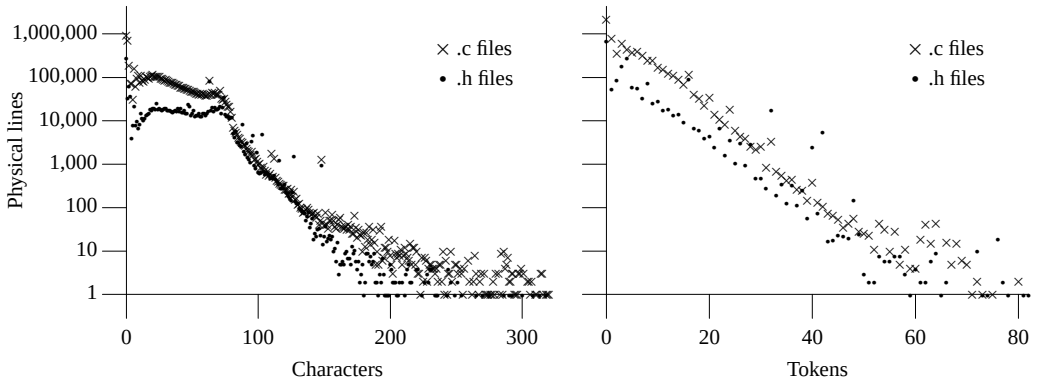


Figure 770.17: Number of physical lines containing a given number of non-white-space characters and tokens. Based on the visible form of the `.c` and `.h` files.

Usage

Table 770.3 shows the relative frequency of the different kinds of tokens in a source file (actual token count information is given elsewhere). Adding the percentages for *Preceded by Space* and *First on Line* (or followed by space and last on line) does not yield 100% because of other characters occurring in those positions. Some tokens occur frequently, but contribute a small percentage of the characters in the visible source (e.g., punctuators). Identifier tokens contribute more than 40% of the characters in the `.c` files, but only represent 28.5% of the tokens in those files.

124 translation phase 3

777 preprocessing tokens white space separation

A more detailed analysis of spacing between individual punctuators is given elsewhere.

Table 770.3: Occurrence of kinds of tokens in the visible form of the `.c` and `.h` files as a percentage of all tokens (value in parenthesis is the percentage of all non-white-space characters contained in those tokens), percentage occurrence (for `.c` files only) of token kind where it was preceded/followed by a space character, or starts/finishes a visible line. While comments are not tokens they are the only other construct that can contain non-white-space characters. While the start of a preprocessing directive contains two tokens, these are generally treated by developers as a single entity.

Token	% of Tokens in `.c` files	% of Tokens in `.h` files	% Preceded by Space	% Followed by Space	% First Token on Line	% Last Token on Line
punctuator	53.5 (11.4)	48.1 (7.5)	27.5	29.7	3.7	25.3
identifier	29.8 (43.4)	20.8 (30.6)	54.9	27.6	1.4	1.2
constant	6.9 (3.8)	21.6 (15.3)	70.3	4.4	0.1	1.6
keyword	6.9 (5.8)	5.4 (4.2)	79.9	82.5	10.3	3.6
comment	1.9 (31.0)	3.4 (40.3)	53.4	2.2	41.2	97.4
string-literal	1.0 (4.6)	0.8 (2.2)	59.9	5.7	0.7	8.0
pp-directive	0.9 (1.1)	4.9 (4.4)	4.7	78.4	0.0	18.2
header-name	0.0 (0.0)	0.0 (0.0)	–	–	–	–

Constraints

771 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.

preprocessing token shall have lexical form

Commentary

The only way a preprocessing token, which does not also have the form of a token, can occur in a strictly conforming program is for it to be either stringized, be the result of preprocessing token gluing, or be skipped as part of a conditional inclusion directive. For instance, the preprocessing token `0.1.1` does not have any of these forms.

1950 # operator 1958 ## operator conditional inclusion

Other Languages

Languages differ in their handling of incorrectly formed tokens, or lexical errors (depending on your point of view). The final consequences are usually the same; the source code is considered to be ill-formed (in some way).

Semantics

A *token* is the minimal lexical element of the language in translation phases 7 and 8. 772

Commentary

A member of the source character set or a preprocessing token is the minimal lexical element of the language in prior translation phases.

translation phases of 115

C++

The C++ Standard makes no such observation.

The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. 773

Commentary

This is a restatement of information given in the Syntax clause. Tokens do not include the preprocessor-token header names. Also the preprocessor-token pp-number is converted to the token constant (provided the conversion is defined). Some of these categories are broken into subcategories. For instance, some identifiers are reserved (they are not formally defined using this term, but they appear in a clause with the title "Reserved identifiers"), and constants might be an *integer-constant*, *floating-constant*, or *character-constant*.

C90

Tokens that were defined to be operators in C90 have been added to the list of punctuators in C99.

C++

2.6p1 *There are five kinds of tokens: identifiers, keywords, literals,[18)] operators, and other separators.*

What C calls constants, C++ calls literals. What C calls punctuators, C++ breaks down into operators and punctuators.

Other Languages

Some languages contain the category of reserved words. These have the form of identifier tokens. They are not part of the language's syntax, but they have a predefined special meaning. Some languages do not distinguish between keywords and identifiers. For instance PL/1, in which it is possible to declare identifiers having the same spelling as a keyword.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. 774

Commentary

Prior to translation phase 3, preprocessing tokens do not exist. The input is manipulated as a sequence of bytes or characters in translation phases 1 and 2.

translation phase 3 124
translation phase 1 116
translation phase 2 118

The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.[58)] 775

Commentary

This is a restatement of information given in the Syntax clause.

C++

In clause 2.4p2, apart from changes to the terminology, the wording is identical.

Other Languages

Many languages do not specify a mechanism for including other source files. However, it is a common extension for implementations of these languages to provide this functionality. So while language specifications do not usually define a category of token called *header names*, they often exist (although implementations very rarely provide a formal language category for this extension).

---

776 If a ' or a " character matches the last category, the behavior is undefined.

character ' or " matches

Commentary

Character constant or string literal tokens cannot include source file new-line characters. A single occurrence of a ' or " character on a logical source line might be said to fall into the category of *non-white-space character that cannot be one of the above*. However, the Committee recognized that many translators treat these two characters in a special way; they tend to simply gather up characters until a matching, corresponding closing quote is found. On reaching a new-line character, having failed to find a matching character, many existing translators would find it difficult to push back, into the input stream, all of the characters that had been read (the behavior necessary to create a preprocessing token consists of a quote character, leaving the following character available for further lexical processing). So the Committee made this as a special case.

In translation phase 7 there is no token into which the preprocessing single non-white-space token could be converted. However, prior to that phase, the token could be stringized during preprocessing; so, this specification could be thought of as being redundant. While an occurrence of this construct may be specified as resulting undefined behavior, all implementations known to your author issue some form of diagnostic when they encounter it.

136 translation phase 7
1950 # operator

One consequence of escape sequences being specified as part of the language syntax is that it is possible to have a sequence of characters that appears to be a character constant but is not. For instance:

866 escape sequence syntax
866 character constant syntax

```
1  char glob = '\d';
2  /*
3   * Tokenizes as: {char}{glob}{=}{'}{\}{d}{'}{;}
4   */
```

Other Languages

A single, unmatched ' or " character is not usually treated as an acceptable token by most languages (which often do not allow a string literal to be split across a source line).

Common Implementations

Most translators issue a diagnostic if an unmatched ' or " character is encountered on a line.

Coding Guidelines

This construct serves no useful purpose and neither would a guideline recommending against using it.

Example

```
1  #define mkstr(a) # a
2
3  char *p = mkstr('); /* Undefined behavior. */
```

---

777 Preprocessing tokens can be separated by *white space*;

preprocessing tokens white space separation

Commentary

white-space within preprocessing directive 1864
translation phase 4 129

C permits a relatively free formatting of the source code. Although there are a few limitations on what kinds of white space can occur in some contexts. Some preprocessing tokens do need to be separated by white space (e.g., an adjacent *pp-number* and *identifier*) for them to be distinguished as two separate tokens. Some preprocessing tokens do not need to be separated by white space to be distinguished (e.g., a punctuator and an identifier). White space can be significant during translation phase 4.

Preprocessing tokens are often separated by significantly more white space than the minimum required to visually differentiate them from each other. This usage of white space is associated with how developers view source code and is discussed in coding guidelines sections.

Other Languages

Fortran spaces not significant

The lexical handling of white space in C is very similar to that found in many other languages. Some languages even allow white space to separate the characters making up a single token. In Fortran white space is not significant. It can occur anywhere between the characters of a token. The most well-known example is:

```
1   DO100I=1,10
```

Here the Fortran lexer has to scan all the characters up to the comma before knowing that this is a do loop (`I` taking on values 1 through 10, the end of the loop being indicated by the label 100) and not an assignment statement:

```
1   DO 100 I=1.10
```

which assigns `1.10` to the variable `DO100I`.

Common Implementations

The time spent performing the lexical analysis needed to create preprocessing tokens is usually proportional to the number of characters in the source file. Many implementations make use of the observation that the first preprocessing token on a line is often preceded by more than one white-space character to perform a special case optimization in the lexer; after reading a new line of characters from the source file, any initial white-space characters are skipped over. (There are a few special cases where this optimization cannot be performed.)

Coding Guidelines

comment /* 934
words white space between 770
expression visual layout 940
declaration visual layout 1348
statement visual layout 1707
translation unit syntax 1810

Separating preprocessing tokens using white space is more than a curiosity or technical necessity (in a few cases). Experience has shown that white space can be used to make it easier for developers to recognize source code constructs. While optimal use of white space for this purpose may be the biggest single discussion topic among developers (commenting practices may be bigger), very little research has been carried out. Issues involving white space between preprocessing tokens are discussed in various subsections, including tokens, expressions, declarations, statements, and translation units.

Example

Some preprocessing tokens require white space:

```
1   typedef int I; /* White space required. */
2
3   I j;           /* White space required. */
4
5   int*p;         /* No white space required. */
6
7   char
8   a[2];          /* new-line is also white space. */
```

Usage

Table 770.3 shows the relative frequency of white space occurring before and after various kinds of tokens.

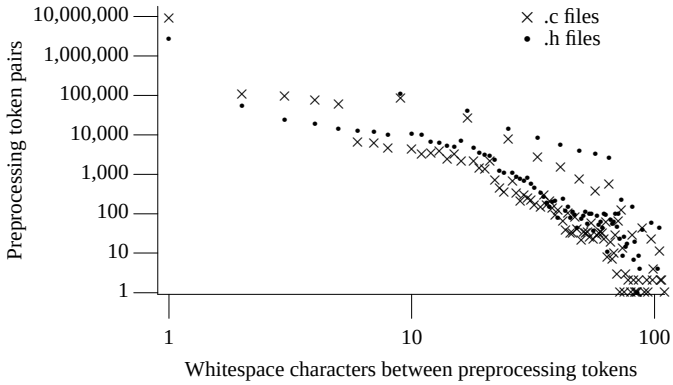


Figure 777.1: Number of *pp-token* pairs having the given number of white-space characters them (does do not include white space at the start of a line— i.e., indentation white space, and end-of-line is not counted as a white-space character). Based on the visible form of the `.c` and `.h` files.

778 this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both.

white-space characters

Commentary

Comments are converted to white space in translation phase 3. The new-line character (white space) causes source code to be displayed in lines. There are no universal character names representing white space.

126 comment replaced by space

Other Languages

New-line is part of the syntax of some languages (e.g., Fortran prior to Fortran 95, Occam, and dialects of Basic). It serves to indicate the end of a statement. C (and C++) has a larger set of characters, which are explicitly defined as white space, than most other languages, which are often silent on the subject.

Coding Guidelines

Technically a comment is treated as a single space character. However, from the point of view of source code visualization, a comment is not a white-space character.

126 comment replaced by space
126 comment replaced by space

Use of horizontal tab is remarkably common (see Table 221.1). The motivation often given by developers for this usage is the desire to minimize the number of characters that they need type (i.e., not having to type multiple space characters). Another reason for this usage is tool based. Many editors support an auto-indentation feature that inserts horizontal tab characters rather than multiple space characters (e.g., `vi`). The visual appearance of the white-space character's horizontal tab, vertical tab, and form-feed depends on implementation-defined characteristics of the device on which the source code is displayed. While use of these characters might incur a cost, if the visual appearance of source code changes when a different display device is used, the benefit of a guideline recommending that they not be used does not appear to be great enough to be worthwhile.

779 As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation.

Commentary

White space between tokens is significant when it appears between tokens that are operands to the # operator.

1952 white space between macro argument tokens

780 White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

white space significant

Commentary

The purpose of white space is to separate preprocessing tokens, resolving ambiguities in some cases, and

786 EXAMPLE

Fortran 777 spaces not significant

to improve the readability of source code. Allowing white space between the characters making up other preprocessing tokens would complicate the job of the lexer, lead to confusion when reading the source code, and does not offer any advantages. A comment is not a preprocessing token, and there is no need to specify the behavior of white space that occurs within it.

Other Languages

Fortran 777 spaces not significant

White space may appear within tokens in some other languages. The space character is not significant in Fortran. All languages that support string literals or character constants allow white space to occur within them.

Common Implementations

The meaning of white space in a header name varies between implementations. Some ignore it, while others, whose hosts support file names containing white space, treat it as part of the file name. Some early translators allowed white space to occur within some preprocessing tokens (e.g., `A = = B;`).

Some translators use different programs to perform different phases of translation. A program that performs preprocessing (translation phases 1–4) is common; its output is written to a text file and read in by a program that handles the subsequent phases. Such a preprocessing program has to ensure that it does not create tokens that did not appear in the original source code. In:

```
#define X -3

int glob = 6-X; /* Expands to 6- -3 */
```

a space character has to be inserted between the `-` character and the macro replacement of `X`. Otherwise the token `--` would be created, not two `-` tokens.

Coding Guidelines

While white space in file names is very easily overlooked, both in a directory listing and in a header name, it rarely occurs in practice.

---

footnote 58

58) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files. 781

Commentary

placemarker 1962 preprocessor

Placemarkers were introduced in C99 as a method of clarifying the operation of the # and ## preprocessor operators. They may or may not exist as preprocessing tokens within a translator. They exist as a concept in the standard and are only visible to the developer through their use in the specification of preprocessor behavior.

C90

The term *placemarker* is new in C99. They are needed to describe the behavior when an empty macro argument is the operand of the ## operator, which could not occur in C90.

C++

This category was added in C99 and does not appear in the C++ Standard, which has specified the preprocessor behavior by copying the words from C90 (with a few changes) rather than providing a reference to the C Standard.

Common Implementations

Implementations vary in how they implement translation phases 3 and 4. Some implementations use a variety of internal preprocessing tokens and flags to signify various events (i.e., recursive macro invocations) and information (i.e., line number information). Such implementation details are invisible to the user of the translator.

---

preprocessing token longest sequence of characters

If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. 782

Commentary

This is sometimes known as the *maximal munch rule*. Without this rule, earlier phases of translation would have to be aware of language syntax, a translation phase 7 issue. The creation of preprocessing tokens (lexing) is independent of syntax in C. maximal munch

Other Languages

A rule similar to this is specified by most language definitions.

Common Implementations

Some implementations use a lexer based on a finite state machine, often automatically produced by a lexical description fed into a tool. These automatically generated lexers have the advantage of being simple to produce, but their ability to recovery from lexical errors in the source code tends to be poor. Many commercial translators use handwritten lexers, where error recover can be handled more flexibly.

While the algorithms described in many textbooks can have a performance that is quadratic on the length of the input,[1156] the lexical grammar used by many programming languages has often been designed to avoid these pathological cases.

Coding Guidelines

Developers do not always read the visible source in a top/down left-to-right order. Having a sequence of characters that, but for this C rule, could be lexed in a number of different ways is likely to require additional cognitive effort and may result in misinterpretations of what has been written. The following guideline also catches those cases where a sequence of three identical characters is read as a sequence of two characters. 770 reading kinds of

Cg 782.1

White space or parentheses shall be used to separate preprocessing tokens in those cases where the tokenization of a sequence of characters would give a different result if the characters were processed from right-to-left (rather than left-to-right).

Example

```
1  extern int ei_1, ei_2, ei_3;
2
3  void f(void)
4  {
5  ei_1 = ei_2 +++ei_3;  /* Incorrectly read as + ++ ? */
6  ei_1 = ei_2--- +ei_3; /* Incorrectly read as -- + ? */
7  }
```

---

783 There is one exception to this rule: ~~a header name preprocessing token is only recognized within a **#include** preprocessing directive, and within such a directive,~~ Header name preprocessing tokens are recognized only within **#include** preprocessing directives or in implementation-defined locations within **#pragma** directives. header name exception to rule

Commentary

This exception means that the following sequence of characters:

```
1  if (a < b && c > d)
```

is lexed as (individual preprocessing tokens are enclosed between matching braces, { }):

```
1  {if} {(} {a} {<} {b} {&&} {c} {>} {d} {)}
```

not as:

```
1   {if} {(} {a} {< b && c >} {d} {)}
```

preprocessing token 782
longest sequence of characters
#pragma 1994 directive

even though the latter meets the longest sequence of characters requirement.

An implementation may chose to support a **#pragma** directive that contains a header name. Such an implementation may need to apply this exception in this context.

The sentence was changed and split in two by the response to DR #324.

C90

This exception was not called out in the C90 Standard and was added by the response to DR #017q39.

C++

This exception was not called out in C90 and neither is it called out in the C++ Standard.

Other Languages

A method of including other source files is not usually defined by other languages. However, implementation of those languages often provide such a construct. The specification given in these cases rarely goes into the same level of detail provided by a language specification. Invariably the behavior is the same as for C—there is special case processing based on context.

Common Implementations

This is what all implementations do. No known implementation looks for header names outside of a **#include** preprocessor directive.

In such contexts, a sequence of characters that could be either a header name or a string literal is recognized as the former. 784

Commentary

header name 918 syntax
string literal 895 syntax
escape sequences 898 string literal

The syntax of header names requires a quote-delimited *q-char-sequence* which, while having the same syntax as a string literal, requires different semantic processing (e.g., no processing of escape sequences is required).

The sentence was split from the previous one by the response to DR #324.

EXAMPLE 1 The program fragment **1Ex** is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens **1** and **Ex** might produce a valid expression (for example, if **Ex** were a macro defined as **+1**). Similarly, the program fragment **1E1** is parsed as a preprocessing number (one that is a valid floating constant token), whether or not **E** is a macro name. 785

Commentary

Standard C specifies a token-based preprocessor. The original K&R preprocessor specification could be interpreted as a token-based or character-based preprocessor. In a character-based preprocessor, wherever a character sequence occurs even within string literals and character constants, if it matches the name of a macro it will be substituted for.

EXAMPLE +++++

EXAMPLE 2 786

The program fragment **x+++++y** is parsed as **x ++ ++ + y**, which violates a constraint on increment operators, even though the parse **x ++ + ++ y** might yield a correct expression.

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5). 787

6.4.1 Keywords

788 *keyword*: one of

auto	**enum**	**restrict**	**unsigned**
break	**extern**	**return**	**void**
case	**float**	**short**	**volatile**
char	**for**	**signed**	**while**
const	**goto**	**sizeof**	**_Bool**
continue	**if**	**static**	**_Complex**
default	**inline**	**struct**	**_Imaginary**
do	**int**	**switch**	
double	**long**	**typedef**	
else	**register**	**union**	

Commentary

The keywords **const** and **volatile** were not in the base document. The identifier **entry** was reserved by the base document but the functionality suggested by its name (Fortran-style multiple entry points into a function) was never introduced into C. [1] base document

The standard specifies, in a footnote, the form that any implementation-defined keywords should take. [490] footnote 28

C90

Support for the keywords **restrict**, **_Bool**, **_Complex**, and **_Imaginary** is new in C99.

C++

The C++ Standard includes the additional keywords:

bool	**mutable**	**this**
catch	**namespace**	**throw**
class	**new**	**true**
const_cast	**operator**	**try**
delete	**private**	**typeid**
dynamic_cast	**protected**	**typename**
explicit	**public**	**using**
export	**reinterpret_cast**	**virtual**
false	**static_cast**	**wchar_t**
friend	**template**	

The C++ Standard does not include the keywords **restrict**, **_Bool**, **_Complex**, and **_Imaginary**. However, identifiers beginning with an underscore followed by an uppercase letter is reserved for use by C++ implementations (17.4.3.1.2p1). So, three of these keywords are not available for use by developers.

In C the identifier `wchar_t` is a typedef name defined in a number of headers; it is not a keyword.

The C99 header **<stdbool.h>** defines macros named `bool`, `true`, `false`. This header is new in C99 and is not one of the ones listed in the C++ Standard as being supported by that language.

Other Languages

Modula-2 requires that all keywords be in uppercase. In languages where case is not significant keywords can appear in a mixture of cases.

Common Implementations

The most commonly seen keyword added by implementations, as an extension, is **asm**. The original K&R specification included **entry** as a keyword; it was reserved for future use.

The processors that tend to be used to host freestanding environments often have a variety of different memory models. Implementation support for these different memory models is often achieved through the use of additional keywords (e.g., **near**, **far**, **huge**, **segment**, and **interrupt**). The C for embedded systems TR defines the keywords **_Accum**, **_Fract**, and **_Sat**.

Embedded C TR 18

Coding Guidelines

One of the techniques used by implementations, for creating language extensions is to define a new keyword. If developers decided to deviate from the guideline recommendation dealing with the use of extensions, some degree of implementation vendor independence is often desired. Some method for reducing the impact of the use of these keywords, on a program's portability, is needed. The following are a number of techniques:

extensions 95.1 cost/benefit

- *Use of macro names.* Here a macro name is defined and this name is used in place of the keyword (which is the macro's body). This works well when there is no additional syntax associated with the keyword and the semantics of a program are unchanged if it is not used. Examples of this type of keyword include **near**, **far** and **huge**.

- *Limiting use of the keyword in source code.* This is possible if the functionality provided by the keyword can be encapsulated in a function that can be called whenever it is required.

- *Conditional compilation.* Littering the source code with conditional compilation directives is really a sign of defeat; it has proven impossible to control the keyword usage.

If there are additional tokens associated with an extension keyword, there are advantages to keeping all of these tokens on the same line. It simplifies the job of stripping them from the source code. Also a number of static analysis tools have an option to ignore all tokens to the end of line when a particular keyword is encountered. (This enables them to parse source containing these syntactic extensions without knowing what the syntax might be.)

Usage

Usage information on preprocessor directives is given elsewhere (see Table 1854.1).

Table 788.1: Occurrence of keywords (as a percentage of all keywords in the respective suffixed file) and occurrence of those keywords as the first and last token on a line (as a percentage of occurrences of the respective keyword; for `.c` files only). Based on the visible form of the `.c` and `.h` files.

Keyword	.c Files	.h Files	% Start of Line	% End of Line	Keyword	.c Files	.h Files	% Start of Line	% End of Line
if	21.46	15.63	93.60	0.00	**const**	0.94	0.80	35.50	0.30
int	11.31	13.40	47.00	5.30	**switch**	0.75	0.77	99.40	0.00
return	10.18	12.23	94.50	0.10	**extern**	0.61	0.71	99.60	0.40
struct	8.10	10.33	38.90	0.30	**register**	0.59	0.64	95.00	0.00
void	6.24	10.27	28.70	18.20	**default**	0.54	0.58	99.90	0.00
static	6.04	8.07	99.80	0.60	**continue**	0.49	0.33	91.30	0.00
char	4.90	5.08	30.50	0.20	**short**	0.38	0.28	16.00	1.00
case	4.67	4.81	97.80	0.00	**enum**	0.20	0.27	73.70	1.80
else	4.62	3.30	70.20	42.20	**do**	0.20	0.25	87.30	21.30
unsigned	4.17	2.58	46.80	0.10	**volatile**	0.18	0.17	50.00	0.00
break	3.77	2.44	91.80	0.00	**float**	0.16	0.17	54.00	0.70
sizeof	2.23	2.24	11.30	0.00	**typedef**	0.15	0.09	99.80	0.00
long	2.23	1.49	10.10	1.70	**double**	0.14	0.08	53.60	3.10
for	2.22	1.06	99.70	0.00	**union**	0.04	0.06	63.30	6.20
while	1.23	0.95	85.20	0.10	**signed**	0.02	0.01	27.20	0.00
goto	1.23	0.89	94.10	0.00	**auto**	0.00	0.00	0.00	0.00

Semantics

789 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise.

Commentary

A translator converts all identifiers with the spelling of a keyword into a keyword token in translation phase 7. This prevents them from being used for any other purpose during or after that phase. Identifiers that have the spelling of a keyword may be defined as macros, however there is a requirement in the library section that such definitions not occur prior to the inclusion of any library header. These identifiers are deleted after translation phase 4.

[136] translation phase 7

[129] translation phase 4

In translation phase 8 it is possible for the name of an externally visible identifier, defined using another language, to have the same spelling as a C keyword. A C function, for instance, might call a Fortran subroutine called xyz. The function xyz in turn calls a Fortran subroutine called `default`. Such a usage does not require a diagnostic to be issued.

Other Languages

Most modern languages also reserve identifiers with the spelling of keywords purely for use as keywords. In the past a variety of methods for distinguishing keywords from identifiers have been adopted by language designers, including:

- *By the context in which they occur (e.g., Fortran and PL/1).* In such languages it is possible to declare an identifier that has the spelling of a keyword and the translator has to deduce the intended interpretation from the context in which it occurs.
- *By typeface* (e.g., Algol 68). In such languages the developer has to specify, when entering the text of a program into an editor, which character sequences are keywords. (Conventions vary on which keys have to be pressed to specify this treatment.) Displays that only support a single font might show keywords in bold, or underline them.

- *Some other form of visually distinguishable feature* (e.g., Algol 68, Simula). This feature might be a character prefix (e.g., **'begin** or **.begin**), a change of case (e.g., keywords always written using uppercase letters), or a prefix and a suffix (e.g., **'begin'**).

The term *stropping* is sometimes applied to the process of distinguishing keywords from identifiers.

Lisp has no keywords, but lots of predefined functions.

In some languages (e.g., Ada, Pascal, and Visual Basic) the spelling of keywords is not case sensitive.

Common Implementations

Linkers are rarely aware of C keywords. The names of library functions, translated from other languages, are unlikely to be an issue.

Coding Guidelines

A library function that has the spelling of a C keyword is not callable directly from C. An interface function, using a different spelling, has to be created. C coding guidelines are unlikely to have any influence over other languages, so there is probably nothing useful that can be said on this subject.

The keyword _Imaginary is reserved for specifying imaginary types.[59)] 790

Commentary

This sentence was added by the response to DR #207. The Committee felt that imaginary types were not consistently specified throughout the standard. The approach taken was one of minimal disturbance, modifying the small amount of existing wording, dealing with these types. Readers are referred to Annex G for the details.

footnote 59

---

59) One possible specification for imaginary types appears in Annex G. 791

Commentary

This footnote was added by the response to DR #207.

6.4.2 Identifiers

6.4.2.1 General

identifier syntax

792

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit
identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters
nondigit: one of
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
digit: one of
0 1 2 3 4 5 6 7 8 9

Commentary

From the developer's point of view identifiers are the most important tokens in the source code. The reasons for this are discussed in the Coding guidelines section that follows.

C90

Support for *universal-character-name* and "other implementation-defined characters" is new in C99.

C++

The C++ Standard uses the term *nondigit* to denote an *identifier-nondigit*. The C++ Standard does not specify the use of *other implementation-defined characters*. This is because such characters will have been replaced in translation phase 1 and not be visible here.

116 translation phase 1

Other Languages

Some languages do not support the use of underscore, _, in identifiers. There is a growing interest from the users of different computer languages in having support for *universal-character-name* characters in identifiers. But few languages have gotten around to doing anything about it yet. What most other languages call operators can appear in identifiers in Scheme (but not as the first character). Java was the first well-known language to support *universal-character-name* characters in identifiers.

Common Implementations

Some implementations support the use of the $ character in identifiers.

Coding Guidelines

1 Introduction

1.1 Overview

identifier introduction

This coding guideline section contains an extended discussion on the issues involved with reader's use of identifier names, or spellings.[792.1] It also provides some recommendations that aim to prevent mistakes from being made in their usage.

Identifiers are the most important token in the visible source code from the program comprehension perspective. They are also the most common token (29% of the visible tokens in the `.c` files, with comma being the second most common at 9.5%), and they represent approximately 40% of all non-white-space characters in the visible source (comments representing 31% of the characters in the `.c` files).

From the developer's point of view, an identifier's spelling has the ability to represent another source of information created by the semantic associations it triggers in their mind. Developers use identifier spellings both as an indexing system (developers often navigate their way around source using identifiers) and as an aid to comprehending source code. From the translators point of view, identifiers are simply a meaningless sequence of characters that occur during the early stages of processing a source file. (The only operation it needs to be able to perform on them is matching identifiers that share the same spellings.)

identifier cue for recall

The information provided by identifier names can operate at all levels of source code construct, from providing helpful clues about the information represented in objects at the level of C expressions (see Figure 792.1) to a means of encapsulating and giving context to a series of statements and declaration in a function definition. An example of the latter is provided by a study by Bransford and Johnson[153] who

[792.1] Common usage is for the character sequence denoting an identifier to be called its *name*; these coding guidelines often use the term *spelling* to prevent possible confusion.

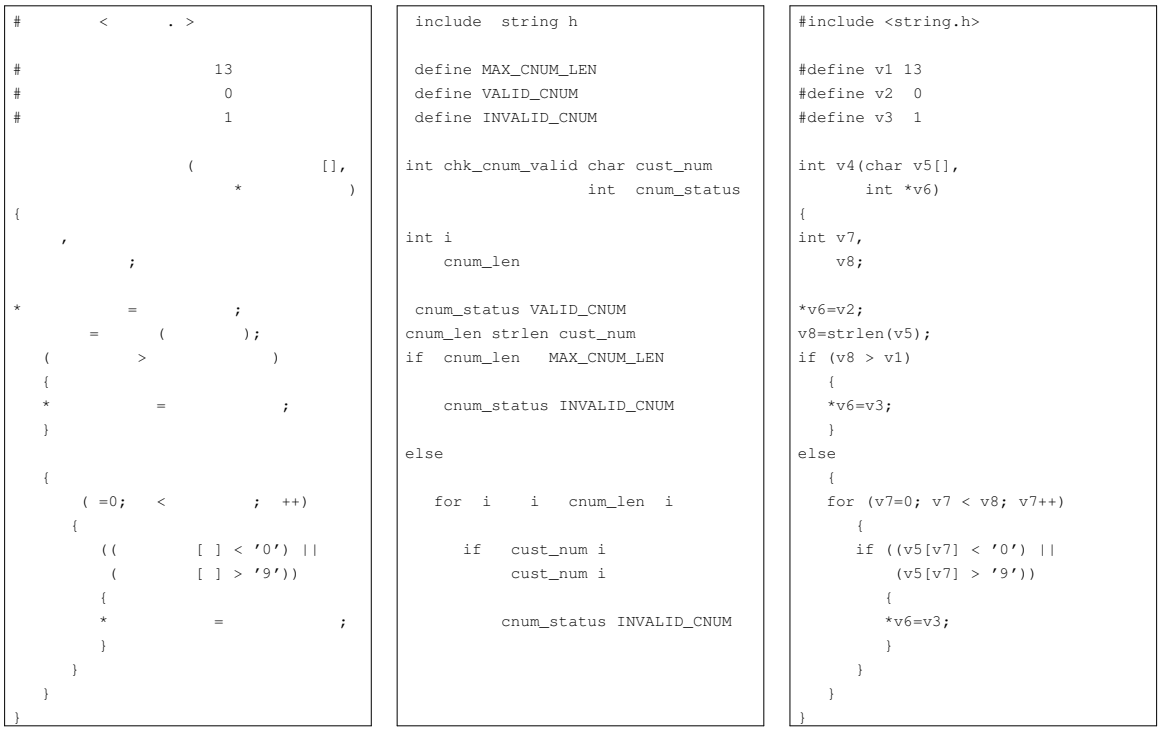


Figure 792.1: The same program visually presented in three different ways; illustrating how a reader's existing knowledge of words can provide a significant benefit in comprehending source code. By comparison, all the other tokens combined provide relatively little information. Based on an example from Laitinen.[795]

read subjects the following passage (having told them they would have to rate their comprehension of it and would be tested on its contents).

Bransford and Johnson[153]

The procedure is really quite simple. First you arrange things into different groups depending on their makeup. Of course, one pile may be sufficient depending on how much there is to do. If you have to go somewhere else due to lack of facilities that is the next step, otherwise you are pretty well set. It is important not to overdo any particular endeavor. That is, it is better to do too few things at once than too many. In the short run this may not seem important, but complications from doing too many can easily arise. A mistake can be expensive as well. The manipulation of the appropriate mechanisms should be self-explanatory, and we need not dwell on it here. At first the whole procedure will seem complicated. Soon, however, it will become just another facet of life. It is difficult to foresee any end to this task in the immediate future, but then one never can tell.

Table 792.1: Mean comprehension rating and mean number of ideas recalled from passage (standard deviation is given in parentheses). Adapted from Bransford and Johnson.[153]

	No Topic Given	Topic Given After	Topic Given Before	Maximum Score
Comprehension	2.29 (0.22)	2.12 (0.26)	4.50 (0.49)	7
Recall	2.82 (0.60)	2.65 (0.53)	5.83 (0.49)	18

The results (see Table 792.1) show that subjects recalled over twice as much information if they were given a meaningful phrase (the topic) before hearing the passage. The topic of the passage describes washing clothes.

The basis for this discussion is human language and the cultural conventions that go with its usage. People spend a large percentage of their waking day, from an early age, using this language (in spoken and written

form). The result of this extensive experience is that individuals become tuned to the commonly occurring sound and character patterns they encounter (this is what enables them to process such material automatically without apparent effort). This experience also results in an extensive semantic network of associations for the words of a language being created in their head. By comparison, experience reading source code pales into insignificance.

770 reading practice
0 automatization
792 semantic networks

These coding guidelines do not seek to change the habits formed as a result of this communication experience using natural language, but rather to recognize and make use of them. While C source code is a written, not a spoken language, developers' primary experience is with a spoken language that also has a written form.

The primary factor affecting the performance of a person's character sequence handling ability appears to be the characteristics of their native language (which in turn may have been tuned to the operating characteristics of its speakers' brain[334]). This coding guideline discussion makes the assumption that developers will attempt to process C language identifiers in the same way as the words and phrases of their native language (i.e., the characteristics of a developer's native language are the most significant factor in their processing of identifiers; one study[763] was able to predict the native language of non-native English speakers, with 80% accuracy, based on the text of English essays they had written). The operating characteristics of the brain also affect performance (e.g., short-term memory is primarily sound based and information lookup is via spreading activation).

There are too many permutations and combinations of possible developer experiences for it to be possible to make general recommendations on how to optimize the selection of identifier spellings. A coding guideline recommending that identifier spellings match the characteristics, spoken as well as written, and conventions (e.g., word order) of the developers' native language is not considered to be worthwhile because it is a practice that developers appear to already, implicitly follow. (Some suggestions on spelling usage are given.) However, it is possible to make guideline recommendations about the use of identifier spellings that are likely to be a cause of problems. These recommendations are essentially filters of spellings that have already been chosen.

792 identifier suggestions
792 identifier filtering spellings

The frequency distribution of identifiers is characterised by large numbers of rare names. One consequence of this is some unusual statistical properties, e.g., the mean frequency changes as the amount of source codes measured increases and relative frequencies obtained from large samples are not completely reliable estimators of the total population probabilities. See Baayen[68] for a discussion of the statistical issues and techniques for handling these kind of distributions.

1.2 Primary identifier spelling issues

There are several ways of dividing up the discussion on identifier spelling issues (see Table 792.2). The headings under which the issues are grouped is a developer-oriented ones (the expected readership for this book rather than a psychological or linguistic one). The following are the primary issue headings used:

identifier primary spelling issues

- *Memorability*. This includes recalling the spelling of an identifier (given some semantic information associated with it), recognizing an identifier from its spelling, and recalling the information associated with an identifier (given its spelling). For instance, what is the name of the object used to hold the current line count, or what information does the object `zip_zap` represent?
- *Confusability*. Any two different identifier spellings will have some degree of commonality. The greater the number of features different identifiers have in common, the greater the probability that a reader will confuse one of them for the other. Minimizing the probability of confusing one identifier with a different one is the ideal, but these coding guidelines attempt have the simpler aim of preventing mutual confusability between two identifiers exceeding a specified level,
- *Usability*. Identifier spellings need to be considered in the context in which they are used. The memorability and confusability discussion treats individual identifiers as the subject of interest, while usability treats identifiers as components of a larger whole (e.g., an expression). Usability factors include the cognitive resources needed to process an identifier and the semantic associations they

expression 940
visual layout

evoke, all in the context in which they occur in the visible source (a more immediate example might be the impact of its length on code layout). Different usability factors are likely to place different demands on the choice of identifier spelling, requiring trade-offs to be made.

Table 792.2: Break down of issues considered applicable to selecting an identifier spelling.

	Visual	Acoustic	Semantic	Miscellaneous
Memory	Idetic memory	Working memory is sound based	Proper names, LTM is semantic based	spelling, cognitive studies, Learning
Confusability	Letter and word shape	Sounds like	Categories, metaphor	Sequence comparison
Usability	Careful reading, visual search	Working memory limits, pronounceability	interpersonal communication, abbreviations	Cognitive resources, typing

memory 0
developer

A spelling that, for a particular identifier, maximizes memorability and usability while minimizing confusability may be achievable, but it is likely that trade-offs will need to be made. For instance, human short-term memory capacity limits suggest that the duration of spoken forms of an identifier's spelling, appearing as operands in an expression, be minimized. However, identifiers that contain several words (increased speaking time), or rarely used words (probably longer words taking longer to speak), are likely to invoke more semantic associations in the readers mind (perhaps reducing the total effort needed to comprehend the source compared to an identifier having a shorter spoken form).

If asked, developers will often describe an identifier spelling as being either *good* or *bad*. This coding guideline subsection does not measure the quality of an identifier's spelling in isolation, but relative to the other identifiers in a program's source code.

1.2.1 Reader language and culture

developer
language and
culture

During the lifetime of a program, its source code will often be worked on by developers having different first languages (their native, or mother tongue). While many developers communicate using English, it is not always their first language. It is likely that there are native speakers of every major human language writing C source code.

If English was good enough for Jesus, it is good enough for me (attributed to various U.S. politicians).

Of the 3,000 to 6,000 languages spoken on Earth today, only 12 are spoken by 100 million or more people (see Table 792.3). The availability of cheaper labour outside of the industrialized nations is slowly shifting developers' native language away from those nations' languages to Mandarin Chinese, Hindi/Urdu, and Russian.

Table 792.3: Estimates of the number of speakers each language (figures include both native and nonnative speakers of the language; adapted from Ethnologue volume I, SIL International). Note: Hindi and Urdu are essentially the same language, Hindustani. As the official language of Pakistan, it is written right-to-left in a modified Arabic script and called Urdu (106 million speakers). As the official language of India, it is written left-to-right in the Devanagari script and called Hindi (469 million speakers).

Rank	Language	Speakers (millions)	Writing direction	Preferred word order
1	Mandarin Chinese	1,075	left-to-right also top-down	SVO
2	Hindi/Urdu	575	see note	see note
3	English	514	left-to-right	SVO
4	Spanish	425	left-to-right	SVO
5	Russian	275	left-to-right	SVO
6	Arabic	256	right-to-left	VSO
7	Bengali	215	left-to-right	SOV
8	Portuguese	194	left-to-right	SVO
9	Malay/Indonesian	176	left-to-right	SVO
10	French	129	left-to-right	SVO
11	German	128	left-to-right	SOV
12	Japanese	126	left-to-right	SOV

If, as claimed here, the characteristics of a developer's native language are the most significant factor in their processing of identifiers, then a developer's first language should be a primary factor in this discussion. However, most of the relevant studies that have been performed used native-English speakers as subjects.[792.2] Consequently, it is not possible to reliably make any claims about the accuracy of applying existing models of visual word processing to non-English languages.

The solution adopted here is to attempt to be natural-language independent, while recognizing that most of the studies whose results are quoted used native-English speakers. Readers need to bear in mind that it is likely that some of the concerns discussed do not apply to other languages and that other languages will have concerns that are not discussed.

1.3 How do developers interact with identifiers?

The reasons for looking at source code do not always require that it be read like a book. Based on the various reasons developers have for looking at source the following list of identifier-specific interactions are considered:

identifier
developer
interaction
770 reading
kinds of

- When quickly skimming the source to get a general idea of what it does, identifier names should suggest to the viewer, without requiring significant effort, what they are intended to denote.
- When searching the source, identifiers should not disrupt the flow (e.g., by being extremely long or easily confused with other identifiers that are likely to be seen).
- When performing a detailed code reading, identifiers are part of a larger whole and their names should not get in the way of developers' appreciation of the larger picture (e.g., by requiring disproportionate cognitive resources).
- Trust based usage. In some situations readers extract what they consider to be sufficiently reliable information about an identifier from its spelling or the context in which it is referenced; they do not invest in obtaining more reliable information (e.g., by, locating and reading the identifiers' declaration).

trust based usage

Developers rarely interact with isolated identifiers (a function call with no arguments might be considered to be one such case). For instance, within an expression an identifier is often paired with another identifier (as the operand of a binary operator) and a declaration often declares a list of identifiers (which may, or may not, have associations with each other).

However well selected an identifier spelling might be, it cannot be expected to change the way a reader chooses to read the source. For instance, a reader might keep identifier information in working memory, repeatedly looking at its definition to refresh the information; rather like a person repeatedly looking at their watch because they continually perform some action that causes them to forget the time and don't invest (perhaps because of an unconscious cost/benefit analysis) the cognitive resources needed to better integrate the time into their current situation.

Introducing a new identifier spelling will rarely causes the spelling of any other identifier in the source to be changed. While the words of natural languages, in spoken and written form, evolve over years, experience shows that the spelling of identifiers within existing source code rarely changes. There is no perceived cost/benefit driving a need to make changes.

An assumption that underlies the coding guideline discussions in this book is that developers implicitly, and perhaps explicitly, make cost/accuracy trade-offs when working with source code. These trade-offs also occur in their interaction with identifiers.

0 cost/accuracy
trade-off

1.4 Visual word recognition

This section briefly summarizes those factors that are known to affect visual word recognition and some of the models of human word recognition that have been proposed. A word is said to be recognized when its representation is uniquely accessed in the reader's lexicon. Some of the material in this subsection is based on chapter 6 of *The Psychology of Language* by T. Harley.[542]

word
visual recognition

[792.2] So researchers have told your author, who, being an English monoglot, has no choice but to believe them.

Reading is a recent (last few thousand years) development in human history. Widespread literacy is even more recent (under 100 years). There has been insufficient time for the impact of comparative reading skills to have had any impact on our evolution, assuming that it has any impact. (It is not known if there is any correlation between reading skill and likelihood of passing on genes to future generation.) Without evolutionary pressure to create specialized visual word-recognition systems, the human word-recognition system must make use of cognitive processes designed for other purposes. Studies suggest that word recognition is distinct from object recognition and specialized processes, such as face recognition. A model that might be said to mimic the letter- and word-recognition processes in the brain is the Interactive Activation Model.[910]

The psychology studies that include the use of character sequences (in most cases denoting words) are intended to uncover some aspect of the workings of the human mind. While the tasks that subjects are asked to perform are not directly related to source code comprehension, in some cases, it is possible to draw parallels. The commonly used tasks in the studies discussed here include the following:

naming task

- *The naming task.* Here subjects are presented with a word and the time taken to name that word is measured. This involves additional cognitive factors that do not occur during silent reading (e.g., controlling the muscles that produce sounds).

lexical decision task

word non-word effects 792

- *The lexical decision task.* Here subjects are asked to indicate, usually by pressing the appropriate button, whether a sequence of letters is a word or nonword (where a word is a letter sequence that is the accepted representation of a spoken word in their native language).

semantic categorization task

- *The semantic categorization task.* Here subjects are presented with a word and asked to make a semantic decision (e.g., "is *apple* a fruit or a make of a car?").

The following is a list of those factors that have been found to have an effect on visual word recognition. Studies[16,566] investigating the interaction between these factors have found that there are a variety of behaviors, including additive behavior and parallel operation (such as the Stroop effect).

stroop effect 1641

age of acquisition

- *Age of acquisition*. Words learned early in life are named more quickly and accurately than those learned later.[1506] Age of acquisition interacts with frequency in that children tend to learn the more common words first, although there are some exceptions (e.g., *giant* is a low-frequency word that is learned early).
- *Contextual variability*. Some words tend to only occur in certain contexts (low-contextual variability), while others occur in many different contexts (high-contextual variability). For instance, in a study by Steyvers and Malmberg[1299] the words *atom* and *afternoon* occurred equally often; however, *atom* occurred in 354 different text samples while *afternoon* occurred in 1,025. This study found that words having high-contextual variability were more difficult to recognize than those having low-contextual variability (for the same total frequency of occurrence).
- *Form-based priming* (also known as *orthographic priming*). The form of a word might be thought to have a priming effect; for instance, *CONTRAST* shares the same initial six letters with *CONTRACT*. However, studies have failed to find any measurable effects.

illusory conjunctions

- *Illusory conjunctions*. These occur when words are presented almost simultaneously, as might happen when a developer is repeatedly paging through source on a display device; for instance, the letter sequences *psychment* and *departology* being read as *psychology* and *department*.
- *Length effects*. There are several ways of measuring the length of a word; they tend to correlate with each other (e.g., the number of characters vs. number of syllables). Studies have shown that there is some effect on naming for words with five or more letters. Naming time also increases as the number of syllables in a word increases (also true for naming pictures of objects and numbers with more syllables). Some of this additional time includes preparing to voice the syllables.

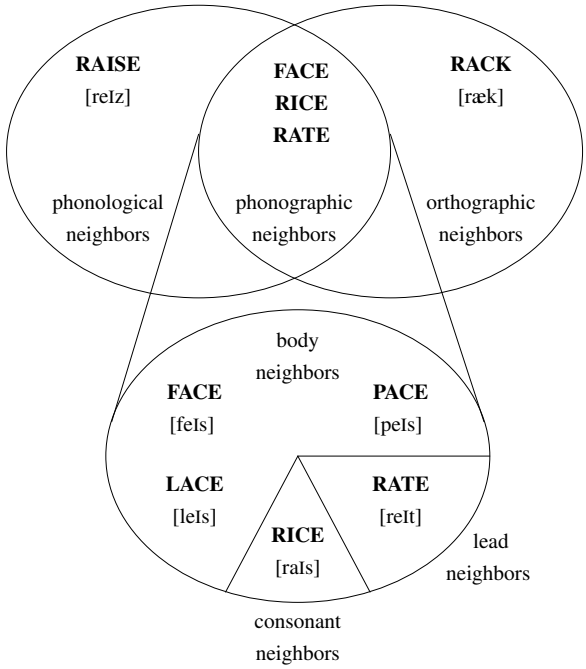


Figure 792.2: Example of the different kinds of lexical neighborhoods for the English word *RACE*. Adapted from Peereman and Content.[1068]

- *Morphology*. The stem-only model of word storage[1327] proposed that word stems are stored in memory, along with a list of rules for prefixes (e.g., *re* for performing something again) and suffixes (*ed* for the past tense), and their exceptions. The model requires that these affixes always be removed before lookup (of the stripped word). Recognition of words that look like they have a prefix (e.g., *interest*, *result*), but don't, has been found to take longer than words having no obvious prefix (e.g., *crucial*). Actual performance has been found to vary between different affixes. It is thought that failure to match the letter sequence without the prefix causes a reanalysis of the original word, which then succeeds. See Vannest[1410] for an overview and recent experimental results.

  morphology identifier

- *Neighborhood effects*. Words that differ by a single letter are known as *orthographic neighbors*. Some words have many orthographic neighbors— *mine* has 29 (*pine*, *line*, *mane*, etc.)— while others have few. Both the *density* of orthographic neighbors (how many there are) and their relative frequency (if a neighbor occurs more or less frequently in written texts) can affect visual word recognition. The *spread* of the neighbors for a particular word is the number of different letter positions that can be changed to yield a neighbor (e.g., *clue* has a spread of two— *glue* and *club*). The rime of neighbors can also be important; see Andrews[38] for a review.

  neighborhood identifier

- *Nonword conversion effect*. A nonword is sometimes read as a word whose spelling it closely resembles.[1113] This effect is often seen in a semantic priming context (e.g., when proofreading prose).

- *Other factors*. Some that have been suggested to have an effect on word recognition include meaningfulness, concreteness, emotionality, and pronounceability,

- *Phonological neighborhood*. Phonological neighborhood size has not been found to be a significant factor in processing of English words. However, the Japanese lexicon contains many homophones. For instance, there are many words pronounced as */kouen/* (i.e., park, lecture, support, etc.). To discriminate homophones, Japanese readers depend on orthographic information (different Kanji compounds). A study by Kawakami[715] showed that phonological neighborhood size affected subjects' lexical decision response time for words written in Katakana.

  phonological neighborhood identifier

  792 phonology

words 792 English

- *Proper names*. A number of recent studies[586] have suggested that the cognitive processing of various kinds of proper names (e.g., people's names and names of landmarks) is different from other word categories.
- *Repetition priming*. A word is identified more rapidly, and more accurately, on its second and subsequent occurrences than on its first occurrence. Repetition priming interacts with frequency in that the effect is stronger for low-frequency words than high-frequency ones. It is also affected by the number of items intervening between occurrences. It has been found to decay smoothly over the first three items for words, and one item for nonwords to a stable long-term value.[919]

semantic priming

- *Semantic priming*. Recognition of a word is faster if it is immediately preceded by a word that has a semantically similar meaning;[1093] for instance, *doctor* preceded by the word *nurse*. The extent to which priming occurs depends on the extent to which word pairs are related, the frequency of the words, the age of the person, and individual differences,
- *Sentence context*. The sentence "It is important to brush your teeth every" aids the recognition of the word *day*, the highly predictable ending, but not *year* which is not.

syllable frequency

- *Syllable frequency*. There has been a great deal of argument on the role played by syllables in word recognition. Many of the empirical findings against the role of syllables have been in studies using English; however, English is a language that has ambiguous and ill-defined syllable boundaries. Other languages, such as Spanish, have well-defined syllable boundaries. A study by Álvarev, Carreiras, and de Vega[22] using Spanish-speaking subjects found that syllable frequency played a much bigger role in word recognition than in English.

word frequency

- *Word frequency*. The number of times a person has been exposed to a word effects performance in a number of ways. High-frequency words tend to be recalled better, while low-frequency words tend to be better recognized (it is thought that this behavior may be caused by uncommon words having more distinctive features,[890, 1228] or because they occur in fewer contexts[1299]). It has also been shown[567] that the attentional demands of a word-recognition task are greater for less frequent words. Accurate counts of the number of exposures an individual has had to a particular word are not available, so word-frequency measures are based on counts of their occurrence in large bodies of text. The so-called *Brown* corpus[780] is one well-known, and widely used, collection of English usage. (Although it is relatively small, one million words, by modern standards and its continued use has been questioned.[182]) The British National Corpus[823] (BNC) is more up-to-date (the second version was released in 2001) and contains more words (100 million words of spoken and written British English).

word non-word effects 792

- *Word/nonword effects*. Known words are responded to faster than nonwords. Nonwords whose letter sequence does not follow the frequency distribution of the native language are rejected more slowly than nonwords that do.

1.4.1 Models of word recognition

Word recognition models of

Several models have been proposed for describing how words are visually recognized.[660] One of the main issues has been whether orthography (letter sequences) are mapped directly to semantics, or whether they are first mapped to phonology (sound sequences) and from there to semantics. The following discussion uses the Triangle model.[544] (More encompassing models exist; for instance, the Dual Route Cascade model[260] is claimed by its authors to be the most successful of the existing computational models of reading. However, because C is not a spoken language the sophistication and complexity of these models is not required.)

By the time they start to learn to read, children have already built up a large vocabulary of sounds that map to some meaning (*phonology* ⇒ *semantics*). This existing knowledge can be used when learning to read alphabetic scripts such as English (see Siok and Fletcher[1246] for a study involving logographic, Chinese, reading acquisition). They simply have to learn how to map letter sequences to the word sounds they already know (*orthography* ⇒ *phonology* ⇒ *semantics*). The direct mapping of sequences of letters to semantics (*orthography* ⇒ *semantics*) is much more difficult to learn. (This last statement is hotly contested by several

logographic 792

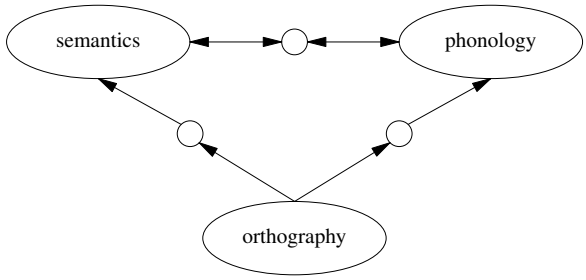


Figure 792.3: Triangle model of word recognition. There are two routes to both semantics and phonology, from orthography. Adapted from Harm.[544]

psychologists and education experts who claim that children would benefit from being taught using the *orthography* ⇒ *semantics* based methods.)

The results of many studies are consistent with the common route, via phonology. However, there are studies, using experienced readers, which have found that in some cases a direct mapping from orthography to semantics occurs. A theory of visual word recognition cannot assume that one route is always used.

The model proposed by[544] is based on a neural network and an appropriate training set. The training set is crucial— it is what distinguishes the relative performance of one reader from another. A person with a college education will have read well over 20 million words by the time they graduate.[792.3]

words domain knowledge

Readers of different natural languages will have been trained on different sets of input. Even the content of courses taken at school can have an effect. A study by Gardner, Rothkopf, Lapan, and Lafferty[472] used 10 engineering, 10 nursing, and 10 law students as subjects. These subjects were asked to indicate whether a letter sequence was a word or a nonword. The words were drawn from a sample of high frequency words (more than 100 per million), medium-frequency (10–99 per million), low-frequency (less than 10 per million), and occupationally related engineering or medical words. The nonwords were created by rearranging letters of existing words while maintaining English rules of pronounceability and orthography.

The results showed engineering subjects could more quickly and accurately identify the words related to engineering (but not medicine). The nursing subjects could more quickly and accurately identify the words related to medicine (but not engineering). The law students showed no response differences for either group of occupationally related words. There were no response differences on identifying nonwords. The performance of the engineering and nursing students on their respective occupational words was almost as good as their performance on the medium-frequency words.

The Gardner et al. study shows that exposure to a particular domain of knowledge can affect a person's recognition performance for specialist words. Whether particular identifier spellings are encountered by individual developers sufficiently often, in C source code, for them to show a learning effect is not known.

2 Selecting an identifier spelling

2.1 Overview

identifier selecting spelling

This section discusses the developer-oriented factors involved in the selection of an identifier's spelling. The approach taken is to look at what developers actually do[792.4] rather than what your author or anybody else thinks they should do. Use of this approach should not be taken to imply that what developers actually do is any better than the alternatives that have been proposed. Given the lack of experimental evidence showing

[792.3] A very conservative reading rate of 200 words per minute, for 30 minutes per day over a 10 years period.

[792.4] Some of the more unusual developer naming practices are more talked about than practiced. For instance, using the names of girl friends or football teams. In the visible form of the `.c` files 1.7% of identifier occurrences have the spelling of an English christian name. However, most of these (e.g., `val`, `max`, `mark`, etc.) have obvious alternative associations. Others require application domain knowledge (e.g., hardware devices: `lance`, floating point `nan`). This leaves a handful, under 0.01%. that may be actual uses of peoples names (e.g., `francis`, `stephen`, `terry`).

that the proposed alternatives live up to the claims made about them, there is no obvious justification for considering them.

Encoding information in an identifier's spelling is generally believed to reduce the effort needed to comprehend source code (by providing useful information to the reader).[792.5]

Some of the attributes, information about which, developers often attempt to encode in an identifier's spelling include:

- *Information on what an identifier denotes.* This information may be application attributes (e.g., the number of characters to display on some output device) or internal program housekeeping attributes (e.g., a loop counter).
- *C language properties of an identifier.* For instance, what is its type, scope, linkage, and kind of identifier (e.g., macro, object, function, etc.).
- *Internal representation information.* What an object's type is, or where its storage is allocated.
- *Management-mandated information.* This may include the name of the file containing the identifier's declaration, the date an identifier was declared, or some indication of the development group that created it.

The encoded information may consist of what is considered to be more than one distinct character sequence. These distinct character sequences may be any combination of words, abbreviations, or acronyms. Joining together words is known as *compounding* and some of the rules used, primarily by native-English speakers, are discussed elsewhere. Studies of how people abbreviate words and the acronyms they create are also discussed elsewhere. Usability issues associated with encoding information about these attributes in an identifier's spelling is discussed elsewhere.

compound 792
word abbreviating 792
identifier identifier 792
encoding usability

optimal spelling identifier

One conclusion to be drawn from the many studies discussed in subsequent sections is that optimal selection of identifier spelling is a complex issue, both theoretically and practically. Optimizing the memorability, confusability, and usability factors discussed earlier requires that the mutual interaction between all of the identifiers in a program's visible source code be taken into account, as well as their interaction with the reader's training and education. Ideally this optimization would be carried out over all the visible identifiers in a programs source code (mathematically this is a constraint-satisfaction problem). In practice not only is constraint satisfaction computationally prohibitive for all but the smallest programs, but adding a new identifier could result in the spellings of existing identifiers changing (because of mutual interaction), and different spelling could be needed for different readers, perhaps something that future development environments will support (e.g., to index different linguistic conventions).

The current knowledge of developer identifier-performance factors is not sufficient to reliably make coding guideline recommendations on how to select an identifier spelling (although some hints are made). However, enough is known about developer mistakes to be able to made some guideline recommendations on identifier spellings that should not be used.

This section treats creating an identifier spelling as a two-stage process, which iterates until one is selected:

1. *A list of candidates is enumerated.* This is one of the few opportunities for creative thinking when writing source code (unfortunately the creative ability of most developers rarely rises above the issue of how to indent code). The process of creating a list of candidates is discussed in the first subsection that follows.
2. *The candidate list is filtered.* If no identifiers remain, go to step 1. The factors controlling how this filtering is performed are discussed in the remaining subsections.

Some of the most influential ideas on how humans communicate meaning using language were proposed by Grice[520] and his maxims have been the starting point for much further research. An up-to-date, easier-

relevance 0

[792.5] The few studies that have investigated this belief have all used inexperienced subjects; there is no reliable experimental evidence to support this belief.

to-follow discussion is provided by Clark,[242] while the issue of relevance is discussed in some detail by Sperber and Wilson.[1271]

More detailed information on the theory and experimental results, which is only briefly mentioned in the succeeding subsections, is provided in the sections that follow this one.

2.2 Creating possible spellings

An assumption that underlies all coding guideline discussions in this book is that developers attempt (implicitly or explicitly) to minimize their own effort. Whether they seek to minimize immediate effort (needed to create the declaration and any associated reference that caused it to be created) or the perceived future effort of using that identifier is not known.

0 cost/accuracy trade-off

Frequency of occurrence of words in spoken languages has been found to be approximately tuned so that shorter ones occur most often. However, from the point of view of resource minimization there is an important difference between words and identifiers. A word has the opportunity to evolve— its pronunciation can change or the concept it denotes can be replaced by another word. An identifier, once declared in the source, rarely has its spelling modified. The cognitive demands of a particular identifier are fixed at the time it is first used in the source (which may be a declaration, or a usage in some context soon followed by a declaration). This point of first usage is the only time when any attempt at resource minimization is likely to occur.

792 Zipf's law

Developers typically decide on a spelling within a few seconds. Selecting identifier spellings is a creative process (one of the few really creative opportunities when working at the source code level) and generates a high cognitive load, something that many people try to avoid. Developers use a variety of cognitive load reducing decision strategies, which include spending little time on the activity.

When do developers create new identifiers? In some cases a new identifier is first used by a developer when its declaration is created. In other cases the first usage is when the identifier is referenced when an expression is created (with its declaration soon following). The semantic associations present in the developer's mind at the time an identifier spelling is selected, may not be the same as those present once more uses of the identifier have occurred (because additional uses may cause the relative importance given to the associated semantic attributes to change).

When a spelling for a new identifier is required a number of techniques can be employed to create one or more possibilities, including the following:

- *Waiting for one to pop into its creator head.* These are hopefully derived from semantic associations (from the attributes associated with the usage of the new identifier) indexing into an existing semantic network in the developers' head.

792 semantic networks

- *Using an algorithm.* For instance, *template* spellings that are used for particular cases (e.g., using `i` or a name ending in `index` for a loop variable), or applying company/development group conventions (discussed elsewhere).

1774 loop control variable
792 identifier other guideline documents

- *Basing the spelling on that of the spellings of existing identifiers* with which the new identifier has some kind of association. For instance, the identifiers may all be enumeration constants or structure members in the same type definition, or they may be function or macro names performing similar operations. Some of the issues (e.g., spelling, semantic, and otherwise) associated with related identifiers are discussed elsewhere.

517 enumeration set of named constants
792 identifier learning a list of

- *Using a tool to automatically generate possibilities for consideration by the developer*. For instance, Dale and Reiter[309] gave a computational interpretation to the Gricean maxims[520] to formulate their *Incremental Algorithm*, which automates the production of *referring expressions* (noun phrases). To be able to generate possible identifiers a tool would need considerable input from the developer on the information to be represented by the spelling. Although word-selection algorithms are used in natural-language generation systems, there are no tools available for identifier selection so this approach is not discussed further here.

822 symbolic name
0 relevance

- *Asking a large number of subjects to generate possible identifier names,* using the most common suggestions as input to a study of subjects' ability to match and recall the identifiers, the identifier having the best match and recall characteristics being chosen. Such a method has been empirically tested on a small example.[78] However, it is much too time-consuming and costly to be considered as a possible technique in these coding guidelines.

Table 792.4: Percentage of identifiers in one program having the same spelling as identifiers occurring in various other programs. First row is the total number of identifiers in the program and the value used to divide the number of shared identifiers in that column). Based on the visible form of the `.c` files.

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql
	46,549	27,467	275,566	52,326	35,868	35,465	18,131
gcc	—	2	9	6	5	3	3
idsoftware	5	—	8	6	5	4	3
linux	1	0	—	1	1	0	0
netscape	5	3	8	—	5	7	3
openafs	6	4	12	8	—	3	5
openMotif	4	3	6	11	3	—	3
postgresql	9	5	12	11	10	6	—

2.2.1 Individual biases and predilections

It is commonly believed by developers that the names they select for identifiers are *obvious*, *self-evident*, or *natural*. Studies of people's performance in creating names for objects shows this belief to be false,[203, 462, 463] at least in one sense. When asked to provide names for various kinds of entities, people have been found to select a wide variety of different names, showing that there is nothing *obvious* about the choice of a name. Whether, given a name, people can reliably and accurately deduce the association intended by its creator is not known (if the results of studies of abbreviation performance are anything to go by, the answer is probably not).

abbreviating 792 identifier

A good naming study example is the one performed by Furnas, Landauer, Gomez, and Dumais,[462, 463] who described operations (e.g., hypothetical text editing commands, categories in *Swap 'n Sale* classified ads, keywords for recipes) to subjects who were not domain experts and asked them to suggest a name for each operation. The results showed that the name selected by one subject was, on average, different from the name selected by 80% to 90% of the other subjects (one experiment included subjects who were domain experts and the results for those subjects were consistent with this performance). The occurrences of the different names chosen tended to follow an inverse law, with a few words occurring frequently and most only rarely.

Zipf's law 792

Individual biases and predilections are a significant factor in the wide variety of names' selection. Another factor is an individual's past experience; there is no guarantee that the same person would select the same name at some point in the future. The issue of general developer difference is discussed elsewhere. The following subsections discuss some of the factors that can affect developers' identifier processing performance.

developer 0 differences

2.2.1.1 Natural language

Developers will have spent significant amounts of time, from an early age, using their native language in both spoken and written forms. This usage represents a significant amount of learning, consequently recognition (e.g., recognizing common sequences of characters) and generation (e.g., creating the commonly occurring sounds) operations will have become automatic.

automatization 0

The following natural-language related issues are discussed in the subsequent sections:

Identifier 792 semantics
Metaphor 792
abbreviating 792 identifier
compound 792 word
identifier 792 English as second language
identifier 792 second language spelling

- Language conventions, including use of metaphors and category formation.
- Abbreviating of known words.
- Methods for creating new words from existing words.
- Second-language usage.

2.2.1.2 Experience

People differ in the experiences they have had. The following are examples of some of the ways in which personal experiences might affect the choice of identifier spellings.

- *recent experience*. Developers will invariably have read source code containing other identifiers just prior to creating a new identifier. A study by Sloman, Harrison, and Malt[1258] investigated how subjects named ambiguous objects immediately after exposure to familiar objects. Subjects were first shown several photographs of two related objects (e.g., chair/stool, plate/bowl, pen/marker). They were then shown a photograph of an object to which either name could apply (image-manipulation software was used to create the picture from photographs of the original objects) and asked to name the object. The results found that subjects tended to use a name consistent with objects previously seen (77% of the time, compared to 50% for random selection; other questions asked as part of the study showed results close to 50% random selection).
- *educational experience*. Although they may have achieved similar educational levels in many subjects, there invariably will be educational differences between developers. A study by Van den Bergh, Vrana, and Eelen[1398] showed subjects two-letter pairs (e.g., *OL* and *IG*) and asked them to select the letter pair they liked the best (for "God knows whatever reason"). Subjects saw nine two-letter pairs. Some of the subjects were skilled typists (could touch type blindfolded and typed an average of at least three hours per week) while the others were not. The letter pair choice was based on the fact that a skilled typist would use the same finger to type both letters of one pair, but different fingers to type the letters of the other pair. Each subject scored 1 if they selected a pair typed with the same finger and 0 otherwise. The expected mean total score for random answers was 4.5. Overall, the typists mean was 3.62 and the nontypists mean was 4.62, indicating that typists preferred combinations typed with different fingers. Another part of the study attempted to find out if subjects could deduce the reasons for their choices; subjects could not. The results of a second experiment showed how letter-pair selection changed with degree of typing skill.
- *cultural experience*. A study by Malt, Sloman, Gennari, Shi, and Wang[892, 893] showed subjects (who were native speakers of either English, Chinese, or Spanish) pictures of objects of various shapes and sizes that might be capable of belonging to either of the categories— bottle, jar, or container. The subjects were asked to name the objects and also to group them by physical qualities. The results found that while speakers of different languages showed substantially different patterns in naming the objects (i.e., a linguistic category), they showed only small differences in their perception of the objects (i.e., a category based on physical attributes).

naming cultural differences

- *environmental experience*. People sometimes find that a change of environment enables them to think about things in different ways. The environment in which people work seems to affect their thoughts. A study by Godden and Baddeley[498] investigated subjects' recall of memorized words in two different environments. Subjects were divers and learned a list of spoken words either while submerged underwater wearing scuba apparatus or while sitting at a table on dry land. Recall of the words occurred under either of the two environments. The results showed that subjects recall performance was significantly better when performed in the same environment as the word list was learned (e.g., both on land or both underwater). Later studies have obtained environmental affects on recall performance in more mundane situations, although some studies have failed to find any significant effect. A study by Fernández and Alonso[40] obtained differences in recall performance for older subjects when the environments were two different rooms, but not for younger subjects.

2.2.1.3 Egotism

It is not uncommon to encounter people's names used as identifiers (e.g., the developer's girlfriend, or favorite film star). While such unimaginative, ego-driven naming practice may be easy to spot, it is possible that much more insidious egotism is occurring. A study by Nuttin[1020] found that a person's name affects

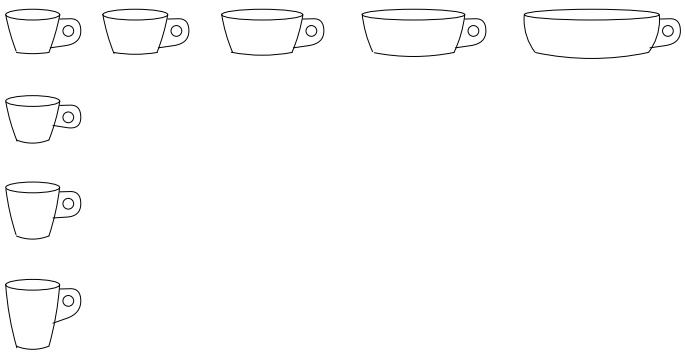


Figure 792.4: Cup- and bowl-like objects of various widths (ratios 1.2, 1.5, 1.9, and 2.5) and heights (ratios 1.2, 1.5, 1.9, and 2.4). Adapted from Labov.[789]

their choice of letters in a selection task. Subjects (in 12 different European countries) were given a sheet containing the letters of their alphabet in random order and spaced out over four lines and asked to circle six letters. They were explicitly told not to think about their choices but to make their selection based on those they felt they preferred. The results showed that the average probability of a letter from the subject's name being one of the six chosen was 0.30, while for non-name letters the probability was 0.20 (there was some variation between languages, for instance: Norwegian 0.35 vs. 0.18 and Finnish 0.35 vs. 0.19). There was some variation across the components of each subject's name, with their initials showing greatest variation and greatest probability of being chosen (except in Norwegian). Nuttin proposed that ownership, in this case a person's name, was a sufficient condition to enhance the likelihood of its component letters being more attractive than other letters. Kitayama and Karasawa[743] replicated the results using Japanese subjects.

A study by Jones, Pelham, Mirenberg, and Hetts[688] showed that the amount of exposure to different letters had some effect on subject's choice. More commonly occurring letters were selected more often than the least commonly occurring (a, e, i, n, s, and t vs. j, k, q, w, x, and z). They also showed that the level of a subject's self-esteem and the extent to which they felt threatened by the situation they were in affected the probability of them selecting a letter from their own name.

2.2.2 Application domain context

context naming affected by

The creation of a name for a new identifier, suggesting a semantically meaningful association with the application domain, can depend on the context in which it occurs.

A study by Labov[789] showed subjects pictures of individual items that could be classified as either cups or bowls (see Figure 792.4). These items were presented in one of two contexts— a neutral context in which the pictures were simply presented and a food context (they were asked to think of the items as being filled with mashed potatoes).

The results show (see Figure 792.5) that as the width of the item seen was increased, an increasing number of subjects classified it as a bowl. By introducing a food context subjects responses shifted towards classifying the item as a bowl at narrower widths.

The same situation can often be viewed from a variety of different points of view (the term *frame* is sometimes used); for instance, commercial events include buying, selling, paying, charging, pricing, costing, spending, and so on. Figure 792.6 shows four ways (i.e., buying, selling, paying, and charging) of looking at the same commercial event.

2.2.3 Source code context

source code context identifier naming conventions identifier 792 other guideline documents hungarian naming identifier

It is quiet common for coding guideline documents to recommend that an identifier's spelling include encoded information on the source code context of its declaration. The term *naming conventions* is often used to refer to these recommendations. Probably the most commonly known of these conventions is the Hungarian naming convention,[1244] which encodes type information and other attributes in the spelling of an identifier.

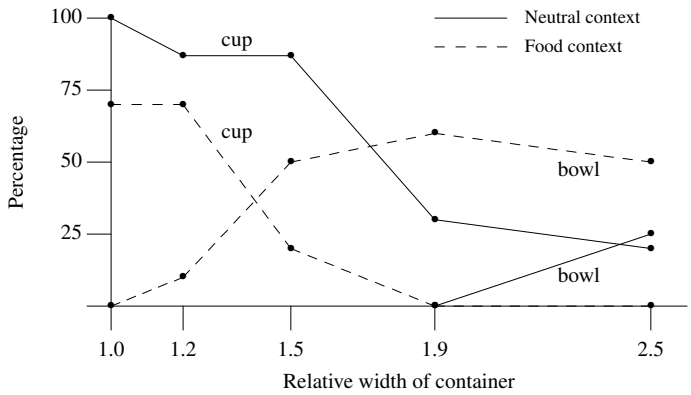


Figure 792.5: The percentage of subjects who selected the term *cup* or *bowl* to describe the object they were shown (the paper did not explain why the figures do not sum to 100%). Adapted from Labov.[789]

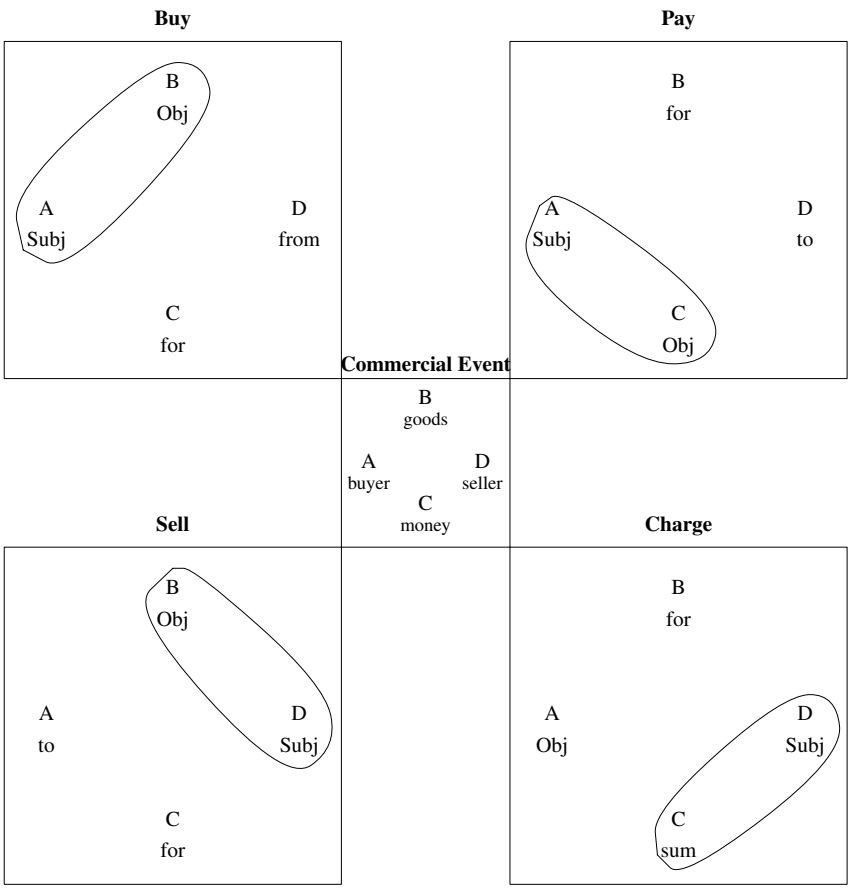


Figure 792.6: A commercial event involving a buyer, seller, money, and goods; as seen from the buy, sell, pay, or charge perspective. Based on Fillmore.[423]

identifier 792 encoding usability

As discussed elsewhere, such information may not be relevant to the reader, may reduce the memorability of the identifier spelling, may increase the probability that it will be confused with other identifiers, and increase the cost of maintaining code.

The two language contexts that are used to influence the spelling of identifiers are namespace and scope. The following subsections briefly discusses some of the issues and existing practices.

2.2.3.1 Name space

macro naming conventions

Macro naming conventions

There is a very commonly used convention of spelling macro names using only uppercase letters (plus underscores and digits; see Table 792.5). Surprisingly this usage does not consume a large percentage of available character combinations (3.4% of all possible four-character identifiers, and a decreasing percentage for identifiers containing greater numbers of characters).

The use of uppercase letters for macro names has become a C idiom. As such, experienced developers are likely to be practiced at recognizing this usage in existing code. It is possible that an occurrence of an identifier containing all uppercase letters that is not a macro name may create an incorrect belief in the mind of readers of the source.

macro parameter naming conventions

preprocessor 1854 directives syntax

There are no common naming conventions based on an identifier being used as a macro parameter. The logical line based nature of macro definitions may result in macro parameter names containing only a few characters having less cost associated with them than those containing many characters.

tag naming conventions

Tag and typedef naming conventions

There is a commonly seen naming convention of giving a tag name and an associated typedef name the same spelling (during the translation of individual translation units of this book's benchmark programs 30% of the tag names declared had the same spelling as that used in the declaration of a typedef name). Sharing the same name has advantage of reducing the amount of information that developers need to remember (once they have learned this convention). As well as this C existing practice, C++ developers often omit the keyword before a tag name (tags are in the same name space as identifiers in C++).

tag 441 name space

syntactic 438 context

Given that one of three keywords immediately precedes a tag name, its status as a tag is immediately obvious to a reader of the source (the only time when this context may not be available is when a tag name occurs as an argument in a macro invocation). Given the immediate availability of this information there is no benefit in a naming convention intended to flag the status of an identifier as a tag.

typedef naming conventions

typedef name 792 no lowercase

The following are several naming conventions that are often seen for typedef names. These include:

- No lowercase letters are used (i.e., uppercase letters, digits, and underscore are used).
- Information on the representation of the type is encoded in the spelling. This encoding can vary from the relatively simply (e.g., `INT_8` indicates that an object is intended to hold values representable in an integer type represented in eight bits; a convention that is consistent with that used in the **`<stdint.h>`** header), or quite complex (e.g., hungarian naming).

MISRA 0 hungarian 792 naming identifier

It is possible for type information, in an identifier's spelling, to be either a benefit or a cost, for readers of the source. For instance, readers may assume that the following equality holds `sizeof(INT_8) == sizeof(char)`, when in fact the author used type **`int`** in the declaration of all `INT_` typedef names.

member naming conventions

Member naming conventions

Some coding guideline documents recommend that the names of members contain a suffix or prefix that denotes their status as members. The cost/benefit of specifying this information in the spelling of an identifier name is discussed elsewhere.

member 443 namespace

label naming conventions

Label naming conventions

There are no common C naming conventions for identifiers that denote labels. However, some coding guideline documents recommend that label names visually draw attention to themselves (e.g., by containing lots of characters). Label name visibility was an explicit goal in the specification of the syntax of labels in Ada. Other coding guideline documents recommend that label names not be visible at all (i.e., they only

labeled 1722 statements syntax

appear within macro replacement lists).

Given that identifiers denoting label names can only occur in two contexts, and no other kinds of identifiers can occur in these contexts, there is no benefit in encoding this information (i.e., is a label) in the spelling. Whether it there is a worthwhile cost/benefit in visually highlighting the use of a label needs to be evaluated on a usage by usage basis. There are a variety of techniques that can be used to provide visual highlighting, it is not necessary to involve an identifier's spelling.

Enumeration constant naming conventions

enumeration constant naming conventions

Some coding guideline documents recommend that the names of members contain a suffix or prefix (e.g., `E_` or `_E`) that denotes their status as members. Unlike member and label names it is not possible to deduce that an identifier is an enumeration constant from the syntactic context in which it occurs. However, there does not appear to be a worthwhile cost/benefit in encoding the status of an identifier as an enumeration constant in its spelling.

The issue of selecting the names of enumeration constants defined in one enumeration type to form a distinct set of symbols is discussed elsewhere.

517 enumeration set of named constants

Function naming conventions

function naming conventions

Some coding guideline documents recommend that the names of functions contain a verb (sometimes a following noun is also specified). A study by Caprile and Tonella[197] created a word grammar describing function names (which was structured in terms of actions) and were able to parse a large percentage of such names in a variety of programs (80% in the case of the mosaic sources).

2.2.3.2 Scope

Tools that automatically generate source code might chose to base part of the spelling of an identifier on its scope to simplify the task of writing the generator. If names followed a fixed unusual, pattern the possibility of duplicates being declared is likely to be reduced.

scope naming conventions

File scope

file scope naming conventions

Some coding guideline documents require identifiers declared in file scope to include a prefix denoting this fact (it is rare to find suffixes being used). The reasons given for this requirement sometimes include issues other than developer readability and memorability; one is management control of globally visible identifiers (exactly why management might be interested in controlling globally visible identifiers is not always clear, but their authority to silence doubters often is).

792 identifier other guideline documents

What are the attributes of an identifier at file scope that might be a consideration in the choice of its name?

- They are likely to be referenced from many function definitions, (unlike block scope identifiers a reader's knowledge of them needs to be retained for longer periods of time).
- They are unlikely to be immediately visible while a developer is looking at source code that references them (unlike block scope identifiers, their declaration is likely to be many lines— hundreds— away from the points of reference).
- They will be unique (unlike block scope names, which can be reused in different function definitions).

During code maintenance new identifiers are often defined at file scope. Does the choice of spelling of these file scope identifiers need to take account of the spelling of all block scope identifiers defined in source files that **`#include`** the header containing the new file scope declaration? The options have the following different costs:

1. Changing the spelling of any block scope identifiers, and references to them, to some other spelling. (This will be necessary if the new, file scope identifier has identical spelling and access to it is required from within the scope in which the local identifier is visible.) There is also the potential cost associated with the block scope identifier not having the ideal attributes, plus the cost of developer relearning associated with the change of an existing identifier spelling.

792 identifier primary spelling issues

2. Selecting another spelling for the file scope identifier. To know that a selected spelling clashes with another identifier requires that the creator of the new identifier have access to all of the source that **#include** the header containing its declaration. There is also the potential cost associated with the file scope identifier not having the ideal attributes. There is no relearning cost because it is a new identifier.

identifier 792 primary spelling issues

3. Accepting the potential cost of deviating from the guideline recommendation dealing with identifier spellings.

Each of these options has different potential benefits; they are, respectively:

identifier 792 primary spelling issues

1. The benefits of following the identifier spelling guideline recommendations are discussed elsewhere. The benefit is deferred.
2. No changes to existing source need to be made, and it is not necessary for developers declaring new file scope identifiers to have access to all of the source that **#include** the header containing its declaration. The benefit is deferred.
3. There is no benefit or immediate cost. There may be a cost to pay later for the guideline deviation.

block scope naming conventions

Block scope

Because of their temporary nature and their limited visibility some coding guideline documents recommend the use of short identifiers (measured in number of characters) for block scope object definitions. What is the rationale for this common recommendation?

typing minimization 0

Some developers openly admit to using short identifiers because they are quicker to type. As pointed out elsewhere, the time taken by a developer to type the characters of an identifier is not significant, compared to the costs to subsequent readers of the source code of a poorly chosen name. Your author suspects that it is the cognitive effort required to create a meaningful name that many developers are really trying to avoid.

What are the properties of identifiers, in block scope, that might be a consideration in the choice of their names?

- They are likely to appear more frequently within the block that defines them than names having file scope (see Figure 1821.5).
- The semantic concepts they denote are likely to occur in other function definitions.
- A program is likely to contain a large number of different block scopes.
- Their length is likely to have greater impact on the layout of the source code than other identifiers.
- Translators do not enforce any uniqueness requirements for names appearing in different block scopes.
- They need to be memorable only while reading the function definition that contains them. Any memories remaining after that block has been read should not cause confusion with names in other function definitions.

2.2.4 Suggestions for spelling usage

identifier suggestions

The following list provide suggestions on how to make the best use of available resources (a reader's mental capabilities) when creating identifier spellings. The studies on which these suggestions are based have mostly used English speakers as subjects. The extent to which they are applicable to developers readers of non-English languages is not known (other suggestions may also be applicable for other languages).

identifiers 792 Greek readers

These suggestions are underpinned by the characteristics of both the written and spoken forms of English and the characteristics of the device used to process character sequences (the human brain). There is likely to be a great deal of interdependence between these two factors. The characteristics of English will have been shaped by the characteristics of the device used to create and process it.

- *Delimiting subcomponents.* Written English separates words with white space. When an identifier spelling is composed of several distinct subcomponents, and it is considered worthwhile to provide a visual aid highlighting them, use of an underscore character between the subcomponents is the closest available approximation to a reader's experience with prose. Some developers capitalize the first letter of each subcomponent. Such usage creates character sequences whose visual appearance are unlike those that readers have been trained on. For this reason additional effort will be needed to process them. In some cases the use of one or more additional characters may increase the effort needed to comprehend constructs containing the identifier (perhaps because of line breaks needed to organize the visible source). Like all identifier spelling decisions a cost/benefit analysis needs to be carried out.

770 words white space between

- *Initial letters.* The start of English words are more significant than the other parts for a number of reasons. The mental lexicon appears to store words by their beginnings and spoken English appears to be optimized for recognizing words from their beginnings. This suggests that it is better to have differences in identifier spelling at the beginning (e.g., *cat*, *bat*, *mat*, and *rat*) than at the end (e.g., *cat*, *cab*, *can*, and *cad*).

initial letters identifier
792 identifier recall
792 words English

- *Pronounceability.* Pronounceability may appear to be an odd choice for a language that is primarily read, not spoken. However, pronounceability is an easy-to-apply method of gauging the extent to which a spelling matches the characteristics of character sequences found in a developers native language. Given a choice, character sequences that are easy to pronounce are preferred to those that are difficult to pronounce.

- *Chunking.* People find it easier to remember a sequence of short (three or four letters or digits) character sequences than one long character sequence. If a non-wordlike character sequence has to be used, breaking the character sequence into smaller chunks by inserting an underscore character between them may be of benefit to readers.

0 memory chunking

- *Semantic associations.* The benefits of identifier spellings that evoke semantic associations, for readers are pointed out in these and other coding guideline documents. However, reliably evoking the desired semantic associations in different readers is very difficult to achieve. Given a choice, an identifier spelling that evokes, in many people, semantic associations related to what the identifier denotes shall be preferred to spellings that evoke them in fewer people or commonly evokes semantic associations unrelated to what the identifier denotes.

- *Word frequency.* High-frequency words are processed more rapidly and accurately than low-frequency words. Given a choice, higher-frequency words are preferred to lower-frequency words.

792 word frequency

2.2.4.1 Existing conventions

In many cases developers are adding identifiers to an existing code base that already contains thousands, if not tens of thousands, of identifiers. The maintainers of this existing code will have learned the conventions used (if only implicitly). Having new identifier spellings follow existing conventions enables maintainers to continue to obtain the benefits from what they have learned, and does not increase costs by requiring that exceptions be handled or new conventions learned. Following an existing convention has a benefit in its own right, independently of the original reasons (which may not even have been valid at the time) for adopting it.

0 implicit learning

One problem with existing source code conventions is finding out what they are. It is possible that the conventions used will vary across the files making up a program (perhaps following the habits and characteristics of the original authors). The following discussion attempts to highlight the main *convention domains* that affect identifier spellings:

- Natural language usage conventions. For instance, word order and creating new words by joining together— *compounding*— existing words. There are often rules for ordering and compounding words and speakers of languages often are sensitive to these rules (usually implicitly following them without consciously being aware of it or even explicit having knowledge of what the rules are).

0 implicit learning

- General software development conventions. For instance, using the abbreviation `ptr` to denote something related to pointers.
- C language developer conventions. For instance, using uppercase letters for identifier spellings denoting macro definitions or typedef names.
- Development group conventions. It is your author's experience that these are rarely effectively enforced and noncompliance is common (even among developers attempting to follow them).[792.6] Without measurements confirming that any development group guidelines are followed in the source being maintained, it is suggested that these conventions be ignored from the point of view of achieving a benefit by considering them when creating new identifier spellings. However, claiming to follow them may reduce the effort of dealing with management, a topic that is outside the scope of this book.
- Host environment conventions. Developers who primarily work within a particular host environment (e.g., Linux or Microsoft Windows) often follow conventions specific to that environment. Whether this is because of the influence of a particular vendor or simply the drifting apart of two communities, is not known.
- An individual's past experience. For all the supposed dynamism of the software business, developers can be remarkably conservative in their identifier-naming habits, often being very resistant to change.

macro 792 naming conventions
typedef 792 naming conventions

Table 792.5: Occurrence of identifier declarations in various scopes and name spaces (as a percentage of all identifiers within the scope/name space in the visible form of the `.c` files; unique identifiers are in parentheses) containing particular character sequences (the phrase *spelled using upper-case letters* is usually taken to mean that no lower-case letters are used, i.e., digits and underscore are included in the possible set of characters; for simplicity and accuracy the set of characters omitted are listed).

	no lower-case	no upper-case	no underscore	no digits	only first character upper-case
file scope objects	0.8 (1.0)	80.3 (79.1)	29.6 (25.4)	87.3 (85.7)	5.2 (5.7)
block scope objects	1.3 (1.8)	91.9 (81.3)	79.9 (58.9)	96.3 (93.0)	1.3 (3.1)
function parameters	0.1 (0.4)	94.2 (82.9)	88.6 (67.4)	96.8 (94.8)	1.4 (2.9)
function definitions	0.2 (0.2)	59.0 (62.1)	27.1 (24.1)	87.1 (86.4)	29.9 (27.3)
struct/union members	0.5 (0.8)	78.5 (71.8)	65.7 (51.3)	93.2 (91.4)	12.0 (14.2)
function declarations	0.7 (0.5)	55.5 (57.1)	27.3 (26.5)	88.7 (87.5)	32.4 (30.1)
tag names	5.7 (6.6)	60.7 (63.8)	25.6 (21.6)	88.1 (85.9)	18.4 (14.5)
typedef names	14.0 (17.0)	37.0 (33.5)	45.0 (40.4)	89.7 (89.3)	39.8 (37.4)
enumeration constants	55.8 (56.0)	10.8 (10.6)	16.0 (15.0)	79.9 (77.9)	32.1 (32.0)
label names	27.2 (48.1)	69.2 (47.4)	70.8 (65.6)	67.4 (46.3)	2.2 (2.3)
macro definitions	78.4 (79.9)	4.9 (5.0)	15.5 (13.0)	70.9 (69.3)	13.1 (11.1)
macro parameters	19.8 (20.4)	77.6 (68.7)	96.0 (83.6)	94.2 (90.7)	1.4 (5.0)

2.2.4.2 Other coding guideline documents

identifier other guideline documents

Other coding guideline documents invariably include recommendations on identifier spelling. These recommendations are sometimes vague to the point of near worthlessness (e.g., "Use meaningful names for identifiers"), or the over-exaggeration of the importance of certain kinds of information with an associated lack of consideration of all the factors involved with identifier usage. Examples of the latter include:

hungarian 792 naming identifier

- *Hungarian notation.* The information provided by this notation is often not needed by the reader, who unnecessarily has to process characters that are essentially noise. This notation also creates maintenance effort in that identifier spellings have to be updated when their types are changed.
- *Meaningful names as brief sentences.* Here priority is given to semantic associations created by an identifiers spelling. The disadvantages of using large numbers of characters in an identifier spelling is discussed elsewhere.

identifier 792 number of characters

[792.6] This experience comes from many onsite visits where a development group's source code was analyzed by a tool configured to enforce that group's identifier-naming conventions.

- *Short names for local identifiers.* Here priority is given to the effort needed to type identifiers and potential use of short-term memory resources (shorter names are likely to require less time to pronounce).
- *Use of prefixes.* Adding prefixes to words has a long history. French scribes in the middle ages would add an *h* to the start of words that were derived from Latin words that started with the letter *h*.[946] The *h* not being pronounced (e.g., modern French *habile* and *honneur*). The introduction of these words into English resulted in either the *h*'s being dropped (*able*), remaining silent (*honour*), or causing a change of pronunciation (*hospital*). The importance of the initial letters, at least for native English speakers, is pointed out above. Mandating the use of prefixes is equivalent to specifying that the information they denote is more important than any other information evoked by an identifiers spelling. If this is true the recommendation to use prefixes is correct, otherwise it causes needless waste of a reader's cognitive resources.
- *No lowercase letters are used (that is, uppercase letters, digits, and underscore are used) for macro definitions and typedef names.* This usage appears to give priority to signaling implementation details to the reader. (While there is a prose text convention, at least in English, that words written using all uppercase letters denote important text, this usage in source code is more of a visual convention than an indication that these kinds of identifiers are more important than others.) Using uppercase letters for macro definitions prevents them from being treated as interchangeable with function definitions (at least for function-like macros). Requiring that macro names be spelled using only uppercase letters creates additional work if the source code is modified. For instance, a macro name that is changed to an enumerated constant or is replaced by an object either has to remain in uppercase or be converted to lowercase. Similarly for a function definition that is remapped to a macro definition (going from lowercase to uppercase). Typedef names appear less often in source code than other kinds of ordinary identifiers. While the syntactic context in which they appear signifies the kind of identifier they represent, readers are likely to be expecting to see a keyword in these contexts (this is the common case). When readers are quickly scanning the source, use of all uppercase letters in the spelling of a typedef name may provide an alternative visual mechanism for rapid recognition (potentially reducing the effort needed to rapidly scan source).

1933 macro function-like

typedef name no lowercase

- *Management/project control.* Here priority is given to information used in the management and coordination of a programs source code. Reserving a set of identifier spellings, usually for future usage, sometimes occurs.

Some coding guideline documents apply identifier-naming conventions that might be applicable in some programming languages (e.g., Cobol[1013]) but are not applicable in C (in the case of Cobol because of the different declaration syntax and in some cases semantics).

2.3 Filtering identifier spelling choices

identifier filtering spellings

This subsection makes guideline recommendations on what identifier spellings should not be used. It does not aim to extensively discuss other spelling filtering issues that developers might consider, although some are covered. It is unlikely to be practical for developers to manually apply these guideline recommendations. Automatic enforcement is assumed to be the most likely method of checking adherence to these recommendations. Whether this automated process occurs at the time an identifier is declared, or sometime later, is a practical cost/benefit issue that is left to the developer to calculate.

The discussion on creating optimal identifier spellings pointed out the need to consider all identifiers declared in the translation of a program. However, the computational cost of considering all identifiers is significant and the guideline recommendations that follow often specify a smaller set of possible identifier spellings that need to be considered.

792 optimal spelling identifier

The basis for these filtering recommendations is the result of the studies described in the major subsections following this one. The major issues are the characteristics of the human mind, the available cognitive resources (which includes a reader's culture and training), and usability factors.

The basic assumption behind the guideline recommendations is that a reduction in similarity between identifiers will result in a reduction in the probability that readers will mistake one for another. The similarity between two identifiers is measured using the typed letters they contain, their visual appearance, and spoken and semantic forms.

2.3.1 Cognitive resources

Other subsections of this coding guideline separate out discussion of issues relating to the functioning of the human brain, and cultural and educational factors. Here they are grouped together as cognitive resources.

An algorithm for calculating the cognitive resources needed to process an identifier spelling is not yet available. For simplicity the following discussion treats each resource as being independent of the others.

2.3.1.1 Memory factors

The primary human memory factors relevant to the filtering of identifier spellings are the limited capacity of short-term memory and its sound-based operating characteristics. The STM capacity limitation issues associated with identifier spelling are discussed elsewhere.

memory 0 developer
identifier 792 STM required

Memory lookup based on the sound sequence representation of a character sequence can sometimes result in information relating to a similar, previously remembered, sound sequence being returned. Also, if two identifiers are both referenced in a related section of source code, it is possible that both of their sound representations will be held in a reader's phonological loop (audio short-term memory) at the same time.

phonological loop 0

The following guideline recommendation is intended to reduce the likelihood of interference, in long and short term memory, between two character sequences that are mapped to similar sound sequences.

Rev 792.1 A newly declared identifier shall have a Levenstein distance, based on the phonemes in its spoken form, of at least two from existing identifiers declared in a program.

This guideline recommendation may be overly restrictive, preventing otherwise acceptable spellings from being used. The following deviation is based on studies showing, at least for native-English speakers, that the start of a word has greater salience than its middle or end.

Dev 792.1 A newly declared identifier may have a Levenstein distance, based on phonemes, of one from existing identifiers declared in a program provided the difference occurs in the first phoneme.

2.3.1.2 Character sequences

Every natural language has patterns in the way in which sounds are joined together to form words. This in turn leads to patterns (at least for nonlogographic orthographies) in the character sequences seen in the written form (even in an irregularly spelled language such as English). Proficient users of a language have overlearned these patterns and can effortlessly recognize them.

logographic 792
orthography 792

While novice readers may read words a letter at a time, experts make use of their knowledge of commonly occurring character sequences (which may be complete words) to increase their reading rate. The penalty for making use of statistical information is an increased likelihood of making mistakes, particularly when reading character sequences that are not required to be words (e.g., identifier spellings).

```
1   enum {OO, O0, l1, ll} glob;
```

Word recognition is driven by both bottom-up processes (the visible characters) and top-down processes (reader expectations). Minimizing the visual similarity between identifier spellings is one technique for reducing the likelihood of a reader mistakenly treating one identifier for another, different, identifier. Although the data needed to calculate an accurate value for the visual similarity between two identifiers is not yet available, the following guideline recommendation is still considered to be worth making.

identifier 792 visual similarity

Cg 792.2 A newly declared identifier shall have a Levenstein distance, based on visual similarity of corresponding characters, of at least two when compared against all identifiers declared in the visible source of a program.

For the purpose of this guideline recommendation, the visual similarity Levenstein distance of two identifiers is defined as the sum, over all pairs of characters, of the visual distance between two characters (one from each identifier) occurring at the same position in the identifier spelling (a space character is used to pad the shorter identifier). The visual distance between two characters is defined as (until a more accurate metric becomes available):

1. zero if they are the same character,
2. zero if one character represents the letter *O* (uppercase oh) and the other is the digit zero,
3. zero if one character represents the letter *l* (lowercase ell) and the other is the digit one,
4. otherwise, one.

2.3.1.3 Semantic associations

The spelling of an identifier is assumed to play a significant role in a readers recall of the semantic information associated with it (another factor is the context in which the identifier occurs). Having two different identifiers with the same spelling and:

identifier semantic associations 438 name space

- with different semantic information associated with them is likely to be create a cost (i.e., recall of information associated with other identifiers sharing the same spelling),
- with the same semantic information associated with them is likely to be create a benefit (i.e., improved recall performance).

0 power law of learning

Cg 792.3 A newly declared identifier shall not have the same spelling as another identifier declared in the same program.

Dev 792.3 A newly declared identifier may have the same spelling as another identifier declared in the same program provided they are both used to denote the same information and both have block scope.

Some identifiers are formed by concatenating two or more known words, abbreviations, or acronyms (these subcomponents are called *conceptual units* here). The interpretation given to a sequence of these conceptual units may not depend significantly on their relative ordering. For instance, either `widget_total` or `total_widget` might be considered to denote a count of the total number of widgets.

The following guideline recommendation is motivated by possible semantic confusion, not by the possibility of typing mistakes. While the general block-edit string matching problem is NP-complete,[868] limiting the comparison to known *conceptual units* significantly reduces the computational cost of checking adherence.

Rev 792.4 A newly declared identifier shall have a Levenstein distance, based on individual conceptual units, of at least two from existing identifiers declared in a program.

Dev 792.4 A newly declared identifier defined in a function definition shall have a Levenstein distance, based on individual conceptual units, of at least two from existing identifiers defined in other function definitions.

In some cases developers may consider two identifiers, differing by a Levenstein distance of one, to be semantically distinct. For instance, `widget_num` or `num_widget` might be considered to denote a number assigned to a particular widget and some count of widgets, respectively. Such an interpretation is dependent on knowledge of English word order and conventions for abbreviating sentences (e.g., "widget number 27" and "number of widgets"). However, this distinction is subtle and relies on too fine a point of interpretation (and could quite easily be given the opposite interpretation) for any form of deviation to be justified.

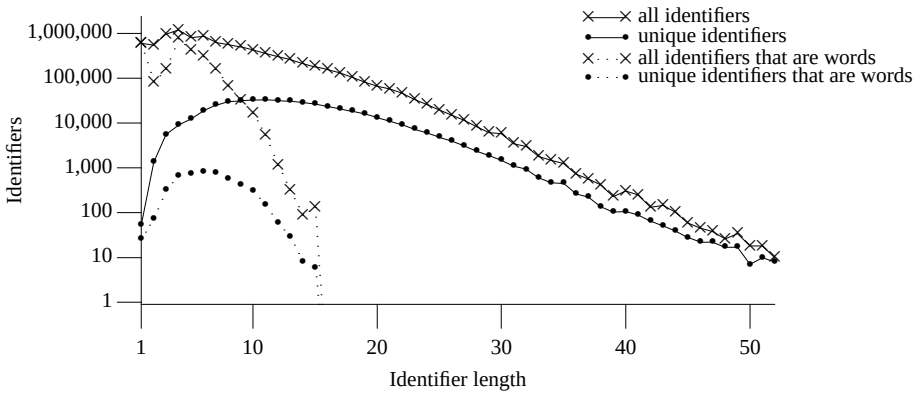


Figure 792.7: Number of identifiers (unique and all) of different length in the visible form of the `.c` files. Any identifier whose spelling appeared in the `aspell` 65,000 word dictionary was considered to be a word.

2.3.2 Usability

identifier encoding usability
identifier 792 usability

The following discussion briefly summarizes the general issues associated with identifier. This issue is discussed in more detail elsewhere.

2.3.2.1 Typing

Developers make mistakes when typing the characters that form an identifier spelling. If two identifier spellings differ by a single character, it is possible that an uncorrected mistake will cause a different identifier to be accessed.

Cg 792.5

A new identifier shall have a Levenstein distance, based on individual characters, of at least two from existing identifiers declared in a program.

An identifier may differ by a Levenstein distance of one from another identifier and not be accessible at the point in the source a typing mistake occurs because it is not visible at that point. Requiring a Levenstein distance of two for all new identifiers may be overly restrictive (preventing otherwise acceptable spellings from being used).

Dev 792.5

An identifier defined in a function definition may have a Levenstein distance, based on individual characters, of one from existing identifiers defined in other function definitions.

2.3.2.2 Number of characters

identifier number of characters
external 283 identifier significant characters
internal 282 identifier significant characters

The C Standard minimum requirements on the number of significant characters in an identifier spelling (the first 31 in an external identifier, and 63 in an internal linkage or macro name) is not usually an issue in human-written code. The issue of translators that have yet to support the new limits (the requirements specified in the previous version of the Standard were lower) are discussed in the limits sentences.

2.3.2.3 Words unfamiliar to non-native speakers

The usual reason for including a word in an identifier spelling is to obtain the benefit of the semantic associations it evokes. If the source is likely to be maintained by developers whose native language is different from that of the author, it is necessary to consider the possibility that some character sequences will not be recognized as words.

Experience shows that technical words often cause the fewest problems. Within a particular application domain, the vocabulary of technical words used at the source code level is usually relatively small (compared to natural languages and even the specific application domain). They are relatively easy for native speakers to identify and L2 speakers may not be embarrassed by their ignorance of these technical terms.

Identifying nontechnical words that may be unfamiliar to non-native speakers is often more difficult. Native speakers rarely have the necessary experience and asking non-native speakers about their language competence may be awkward or impractical.

Although there have been some surveys of L2 vocabulary knowledge,[632] the available information does not appear to be sufficiently extensive to enable a guideline recommendation (that words likely to be unfamiliar to L2 speakers not be used) to be enforced; so none is given here.

2.3.2.4 Another definition of usability

In some cases identifier spelling usability might be defined in terms of satisfying a management requirement other than minimizing future maintenance costs. For instance, the customer may require adherence to a coding guideline document, which recommends that various kinds of semantic information be explicitly encoded in an identifier's spelling (e.g., the MISRA C Guidelines contain an advisory rule that the spellings of typedef names contain information on the number bits in the storage representation of the defined arithmetic type).

0 MISRA

Definitions of usability based on these alternative requirements are not discussed further in this coding guideline section.

3 Human language

This section discusses the characteristics of human languages, primarily in their written form. These empirical and theoretical findings provide background material for the discussion on the three primary requirement issues (memorability, confusability, usability). The section concludes with a more detailed discussion of one language, English.

human language characteristics

3.1 Writing systems

A writing system,[1194] known as an *orthography*, uses written characters to represent the structure of a linguistic system. If the character-to-sound rules are relatively simple and consistent (e.g., German, Spanish, Japanese hirigana), the orthography is said to be *shallow*; while if they are complex and inconsistent (e.g., English), it is said to be *deep*. A writing system is more than a set of characters and the sequences of them used to represent words; there are also the conventions adopted by its writers (e.g., direction of writing and some written abbreviations have no equivalent spoken form).

orthography

Table 792.6: Number of people using particular types of writing system for the top 50 world languages in terms of number of speakers. Literacy rates from UNESCO based on typical countries for each language (e.g., China, Egypt, India, Spain). Adapted from Cook.[273]

Total languages out of 50	Speakers (millions)	Readers (millions, based on illiteracy rates)
Character-based systems— 8 (all Chinese) + Japanese	1,088	930
Syllabic systems— 13 (mostly in India) + Japanese, Korean	561	329
Consonantal systems— 4 (two Arabic) + Urdu, Persian	148	no figures available
Alphabetic systems— 21 (worldwide)	1,572	1,232

Most existing source code is written using only characters from the basic source character set. The introduction of Universal Character Names and growing tool support for extended characters continues to increase the likelihood that developers will encounter identifiers spelled using characters outside of the invariant Latin subset.

221 basic source character set
815 universal character name syntax
215 extended characters
24 ISO 646

There are three kinds of writing systems:

1. *alphabetic.* These writing systems contain a small set of letters. Sequences of one or more of these

grapheme

letters represent the basic spoken units of a word (this sequence of letters is known as a *grapheme* and the sound units it represents is a phoneme) or a complete word. One or more graphemes may be written in sequence to represent a spoken word. This writing system has two varieties:

phoneme 792

- Abjads, or consonant alphabets, predominantly use only consonants in their written forms. Vowels can be added, usually by means of diacritics, but this is not common (Arabic and Hebrew use them in poetry and children's books). Most abjads, with the exception of Divehi hakura and Ugaritic, are written from right-to-left.
- Alphabets, or phonemic alphabets, nearly always represent consonants and vowels in written works (acronyms may not contain any vowels).

Some scripts, for instance Arabic, are used both as an abjad and as an alphabet.

2. *syllabic.* These writing systems use individual characters to represent syllables; for instance, Bengali, Cherokee, and Japanese Katakana.

syllable 792

logographic

3. *logographic.* These writing systems, or logosyllabaries, are the most complex natural language writing systems. They can be broken down into the following:

   - Logograms are symbols that represent whole words, without a phonetic component. Some logograms resemble, or originally resembled, the things they represent and are sometimes known as *pictograms* or *pictographs*.
   - Ideograms are symbols that graphically represent abstract ideas, without a phonetic component.
   - Semantic–phonetic compounds are symbols that include a semantic element, which represents or hints at the meaning of the symbol, and a phonetic element, which denotes or hints at the pronunciation. Some of these compound symbols can be used for their phonetic value alone, without regard for their meaning.

   Examples include Chinese, Japanese Kana, and Ancient Egyptian.

3.1.1 Sequences of familiar characters

When a writing system breaks the representation of a word into smaller components (e.g., alphabetic, phonemic, or syllabic), readers learn not only the visual form of the characters, but also implicitly learn the likelihood of encountering particular character sequences. For instance, readers of English would expect the letter *t* to be followed by *h* and would be surprised to see it followed by *q*. Information on frequency of occurrence of letter sequences in different languages is sufficiently reliable that it is used by cryptographers to help break codes,[470] by OCR software for correcting errors in scanned text, and by mobile phones for predictive text entry.

The frequency of occurrence of particular letter sequences varies between different languages. It will depend on how letter sequences are mapped to phonemes and the frequency of the different phonemes used in the spoken form of a language.

phoneme 792

In his famous paper, *A Mathematical Theory of Communication* Shannon[1220] gave the following example of letter sequences that successively approximated English.

- Zero-order approximation (symbols independent and equiprobable): XFOML RXKHRJFFJUJ ZLP-WCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZLHJQD.
- First-order approximation (symbols independent but with frequencies of English text): OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL.
- Second-order approximation (digram structure of English): ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE.

- Third-order approximation (trigram structure of English): IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME OF DEMONSTURES OF THE REPTAGIN IS REGOACTIONA OF CRE.

The entropy of English has been estimated as 1.75 bits per character[1333] (a random selection from 26 letters or space would have an entropy of 4.75 bits per character). It is this predictability, coupled with a reader's knowledge of it, built up through years of practice, that enables people to process words from their first language much more quickly than words from an unfamiliar language. However, a reader's rate of information extraction does not increase; they simply learn to take account of the redundancy present in the input.

792 identifier information extraction

Information on common letter sequences has been used in models of reader eye movements and typing performance.

770 Mr. Chips
792 typing mistakes

Many of the top 12 languages (see Table 792.3) use letters from the invariant Latin character set (with the exception of English, they also include a few additional characters). Because these languages each use slightly different sets of phonemes and use different letters sequences to represent them, letter sequences that are common in one language may be rare in another.

24 ISO 646

If an identifier spelling contains a word belonging to some natural language, readers unfamiliar with that language may break the character sequence up into different letter sequences than the original author did. The letter sequences may be shorter, possibly a single letter. Some of the issues involved in a reader's handling of unknown character sequences, nonwords, is discussed elsewhere.

792 word pronounceability
792 word nonword effects
792 characters mapping to sound
792 identifier nonword spelling

3.1.2 Sequences of unfamiliar characters

Developers do not need to know anything about any human language, expressed using a particular writing system, to comprehend source code containing instances of that writing system. To comprehend source code readers need to be able to

reading characters unknown to reader

- deduce when two identifiers have the same, or different, spellings. This same/different task only requires the ability to visually compare two identifiers.
- store a representation of some previously seen identifiers (those considered worth remembering). This task requires a mapping from the visual form of the identifier to some internal, in the developer's mind, representation.

How do readers perform when presented with an unfamiliar writing system (e.g., written using universal character names)?

815 universal character name syntax

A study by Brooks[164] investigated subject's performance in identifying character sequences built from characters they were unfamiliar with— for instance, Π*)*(Π*(*. One group of subjects was first taught a character-to-letter mapping (Π⇒*N*, *)*(⇒*E*, Π⇒*A*, *(*)⇒*P*), while the other group was not. Both groups of subjects were asked to learn associations between a number of character sequences and a corresponding spoken response. The spoken form of each character sequence corresponded to the English pronunciation of the word formed by mapping the characters to the letters (which only one group of subjects had been taught). Subject's performance (time taken to speak all character sequences, presented to them in a random order, they had learned) was measured. The results showed an initial performance advantage for subjects using purely visual recognition (they had not been taught a mapping to letters). The performance of both groups of subjects improved with practice. However, given sufficient practice, the performance of subjects who had learned the character to letter mapping exceeded that of subjects who had not learned it. The amount of practice needed seemed to depend on the number of character sequences that had to be learned and their visual form.

A study by Muter and Johns[984] asked subjects (native speakers of English) to learn to identify either a set of logographs (Chinese characters) or words written in an unfamiliar alphabetic code (Devanagari— the written form of Hindi). As expected subjects reaction time and error rates improved with practice. However, the initial performance with logographs was significantly better than alphabetic codes (see Figure 792.8).

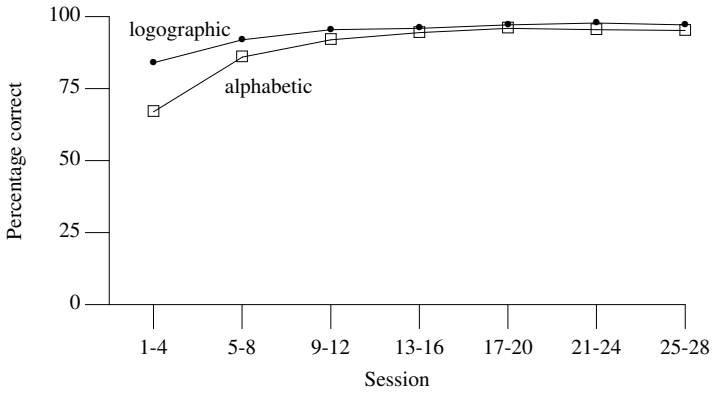


Figure 792.8: Improvement in word-recognition performance with number of sessions (most sessions consisted of 16 blocks of 16 trials). Adapted from Muter and Johns.[984]

letter detection 770

The result of this and other character learning studies shows that people relatively quickly achieve high proficiency. However, previous experience with an alphabetic character set would not seem to confer any advantage during the initial learning phrase of a new set of alphabetic characters.

3.2 Sound system

word sound system

The sounds used by individual natural languages do not use all possible combinations that the human vocal tract is capable of producing.[132] While there may be thousands of possibilities, a particular language usually uses less than a hundred (English uses approximately 44, depending on regional accent).

phoneme

The *phoneme* is the minimal segment of sound that distinguishes one word from another word. These are generally divided into vowels (open sounds, where there are no obstructions to the flow of air from the mouth; known languages contain from 2 to 25 vowels) and consonants (created by some form of obstruction to the passage of air through the speech tract; known languages contain between 5 and more than 100 consonants). Hawaiian contains 12 phonemes, while !Xu (spoken near the Kalahari desert in Southern Africa) has as many as 141. The most commonly encountered vowels, in different languages, are */i/*, */e/*, */a/*, */o/*, */u/*, and the most commonly encountered constants are */p/*, */k/*, */t/*. The vowel */a/* is believed to be the only phoneme that occurs in all languages.

An *allophone* is a phonetically distinct variant of a phoneme. The position of a phoneme within a word can cause it to be pronounced slightly differently; for instance, the */t/* sounds in *hit* and *tip* are allophones. Some languages do not distinguish between different pairs of phonemes; for instance, Japanese treats */l/* and */r/* as being the same phoneme, but different allophones.

phonology

Phonology is the study of spoken sounds (i.e., sequences of phonemes).

Languages contain regularities in the ordering of phonemes. For instance, English words tend to follow a CVC structure (Consonant Vowel Consonant), and even within this structure certain patterns are heard much more frequently than others.[730]

syllable

The next higher-level unit of speech above a segment is known as a *suprasegmental*. The particular suprasegmental of interest to these coding guidelines is the syllable. A *syllable* consists of three parts: (1) the *onset*, (2) the *peak* or *nucleus*, and (3) the *coda*; for instance, for the syllable */man/*, */m/* is the onset, */a/* the peak, and */n/* the coda. A definition of syllable that readers might be familiar with is that of a consonant followed by a vowel (CV); however, this definition cannot be used to syllabify the structure CVCCV— is it CV-CCV or CVC-CV.

English syllables can have zero to three consonants before the vowel and zero to four consonants after the vowel. There has been relatively little research on automatically deducing how to divide a word into its component syllables.[1507] Kessler and Treiman[730] investigated the structure of English syllables and found a correlation between a vowel and its following consonant. However, the correlation between a consonant and its following vowel was significantly lower, meaning the start of an English syllable is more informative

(distinctive) than its ending.

The same syllables can be spoken at various levels of intensity. Varying the intensity, when speaking, enables certain parts of what is being said to be *stressed*. A languages stress pattern is of interest to these coding guidelines because it can affect how words are abbreviated. Different languages stress words in different ways. The difference between stressed and unstressed syllables in English[459] is greater than most other languages and it is common for the odd-numbered syllables to be stressed, with the first receiving the most stress (in prefixed words the primary stress usually falls on the first syllable of the root[212]). The same word may be stressed differently in different languages: English *GRAMmar* (from the French *gramMAIRE*) and *CHOColate* (from the Spanish *chocoLAte*).

word stress
792 abbreviating identifier

The *morpheme* is the smallest grammatical unit of speech. There are two basic types; a *bound morpheme* is defined in terms of how it is attached to the other form, the *free morpheme*. The most common bound morphemes are prefixes and suffixes (e.g., *re-* and *-ed*; see Hawkins and Gilligan[554] for a discussion of prefixing and suffixing universals).

morpheme

The term *morphology* refers to the study of word structure and formation— the syntax of words. Pirkola[1090] discusses the morphology of the world's languages from the perspective of information retrieval. Bauer[100] and Selkirk[1213] discuss English word formation.

morphology

With the exception of logographic writing systems, there is some degree of correspondence between the written and spoken form of a word (spelling is discussed elsewhere). Although C source is not designed to be a spoken language, many developers create a spoken form for it. The norms for this spoken form are most noticeable when they are broken (e.g., self-taught C programmers do not always have the opportunity to listen to C being spoken and invent their own pronunciations for some operators).

792 spelling

3.2.1 Speech errors

Cutler[303] provides a review of speech error data. A systematic, cross-linguistic examination of speech errors in English, Hindi, Japanese, Spanish, and Turkish by Wells-Jensen[1452] found (reformatting a quote from her paper):

- *Languages are equally complex.* No overall differences were found in the numbers of errors made by speakers of the five languages in the study. This supports the basic assumption that no language is more difficult than any other.
- *Languages are processed in similar ways.* Fifteen English-based generalizations about language production were tested to see to what extent they would hold true across languages. It was found that, to a large degree, languages follow similar patterns. For example, all of the languages exhibited the same pattern of semantically-based errors in open-class words, and all exhibited more errors with inflectional than derivational affixes. It was found, however, that the relative numbers of phonological anticipations and perseverations in other languages did not follow the English pattern.
- *Languages differ in that speech errors tend to cluster around loci of complexity within each.* Languages such as Turkish and Spanish, which have more inflectional morphology, exhibit more errors involving inflected forms, while languages such as Japanese, with rich systems of closed-class forms, tend to have more errors involving closed-class items.

3.2.2 Mapping character sequences to sounds

It is believed that there are two possible routes by which readers convert character sequences to sound, (1) memory lookup, a direct mapping from the character sequence to sound; and (2) grapheme-to-phoneme conversion. Irregularly spelled words have to use the first route; for instance, the phrase "pint of water" does not contain any words that rhyme with *mint*. A study by Monsell, Patterson, Graham, Hughes, and Milroy[960] asked subjects to name lists of words, nonwords, or a mixture of both. The results showed that when nonword lists contained a few words, subjects tended to regularize (use the second route) the pronunciation of the words (the error rate was double that for lists containing words only).

characters mapping to sound
792 Word recognition models of

A study by Andrews and Scarratt[39] investigated how people pronounce nonwords. For instance, would subjects pronounce *jead* using the regular grapheme-to-phoneme correspondence heard in *mead*, or would they use the irregular form heard in *head*? The results showed that 90% of pronunciations followed regular grapheme-to-phoneme rules. This is not to say that the pronunciations used for a particular word were always unique. In 63% of cases a nonword received one or two pronunciations, while 9.7% of nonwords received more than four different pronunciations. There seemed to be a common factor for nonwords where an irregular pronunciation was used; these nonwords did not have any regular word-body neighbors. This latter result suggests that perhaps speakers do not use grapheme-to-phoneme rules to build the pronunciations used for nonwords, but rather base it on the pronunciation of words they know that have similar spellings.

neighborhood identifier 792

A study by Gibson, Pick, Osser, and Hammond[487] found that significantly more different pronunciations were used by subjects for letter sequences having a first-order approximation to English than letter sequences having a higher-order approximation. Deshmukh[347] discusses using maximum likelihood estimators to generate (potentially) multiple pronunciations, each with an associated likelihood probability.

Text-to-speech conversion is a very active research area with a growing number of commercial applications. To obtain high-quality output most systems rely on a dictionary of word-to-sound mappings. A number of rule-based algorithms have been proposed for converting letter sequences (graphemes) to sounds (phonemes).

Early work on creating a grapheme-to-phoneme mapping for a language involved a great deal of manual processing. For instance, Berndt, Reggia, and Mitchum[116] manually analyzed 17,310 words to derive probabilities for grapheme-to-phoneme mapping of English. Automating this process has obvious advantages. Daelemans and van den Bosch[307] created a language-independent conversion process that takes a set of examples (words and their corresponding phonetic representation) and automatically creates the grapheme-to-phoneme mapping. More recent research, for instance Pagel, Lenzo, and Black,[1042] has attempted to handle out of vocabulary words (i.e., not in the training set); however, the quality of the conversion varies significantly. An evaluation of publicly available algorithms,[312] using a 16,280 word dictionary, found correct conversion rates of between 25.7% and 71.8%.[792.7]

Divay and Vitale[364] provide a discussion of recent algorithms for grapheme–phoneme conversion of English and French, while Peereman and Content[1069] provide quantitative information on the regularity of the mapping between orthography and phonology in English and French.

A category of letter sequences that often does not have the characteristics of words are people's names, particularly surnames. Correct pronunciation of people names is important in many applications and algorithms have been designed to specifically handle them. What can be learned from attempts to convert the written form of people's names to sound? A comparison of eight name-pronunciation systems[502] (two of which were human) found that acceptable (400-name test set, as judged by a panel of 14 listeners) performance varied from 78% to 98% for high-frequency names to 52% to 96% for very low-frequency names. Many of the systems tested included a pronunciation dictionary of several thousand to handle common cases and to help supplement the rules used (a collegiate-level English dictionary usually lists approximately 250,000 words). A list of unique surnames (in the USA) contains more than 1.5 million entries. To cover 50% of the surnames in the USA requires 2,300 names, while 50% of ordinary words can be covered in 141 words).

Vitale[1425] and Llitjós[862] found that by taking into account the language of origin of proper names (statistical properties of the letter sequences in a name have been found to be a good indicator of its language of origin) it was possible to improve the accuracy of the phoneme transcription.

The Soundex algorithm[588] is often mentioned in textbooks which discuss *sounds-like*, and has been used in a number of applications where a sounds-like capability is needed. This algorithm converts a word to a code consisting of the first letter of the word followed by up to three digits (obtained by looking up a value between zero and seven for subsequent letters). Its very simplistic approach delivers results that are better than might be expected (a 33% success rate, with a 25% failure rate has been found for surnames[233]).

[792.7] Some researchers quote phoneme-conversion accuracy on a per letter-basis, while others use a per phoneme basis. A 90% per letter conversion accuracy equates to a $0.9^6 = 53\%$ word-conversion accuracy (for a six-letter word).

3.3 Words

Words do not exist in isolation; they belong to one or more human languages. Until relatively recently people experienced words in spoken form only. A number of studies[823] have found significant differences between spoken and written language usage. Because of the relatively high cost of analyzing spoken words compared to analyzing (invariably automated, using computers) written words, there has been significantly more published research on the written form. Source code is a written language and the studies quoted in this coding guideline section have primarily been of written text.

Many languages allow new words to be created by joining together, *compounding*, existing words. Other techniques used to create words include *prefixation* (*sleep* ⇒ *asleep*, *war* ⇒ *miniwar*) and *suffixation* (*kitchen* ⇒ *kitchenette*, *sea* ⇒ *seascape*).

In agglutinative languages, such as Japanese, the definition of a word is not always clear-cut. Words can be built up, like beads-on-a-string, using a series of affixes to the *root* word (the word that appears in a dictionary). There may also be letter changes at morpheme for phonetic reasons. For instance,[1028] the Turkish root word *uygar*, meaning civilized, might have suffixes added to build the single word (translated as "(behaving) as if you were one of those whom we not be able to civilize", written with the components separated by + rather being immediately adjacent):

agglutinative languages

792 morpheme

```
uygar+laş+tir+ama+yabil+ecek+ler+imiz+den+miş+siniz+cesine
```

Part of the process of learning a language is producing and understanding compound words. The rules used and the interpretation given varies across languages. A study by Boucher[141] investigated the problems French students have in forming English compound words. He found that students made a number of different mistakes: used more than two terms, used incorrect affixes, pluralized the first word, or applied the French rules. For instance, when asked the word for "a dog which hunts birds", answers included *dog-bird* and *bird-dog-hunting*. A later project by Boucher, Danna, and Sébillot[142] produced an intelligent tutoring system to teach students how to form English compound words.

While there might be general agreement on the pattern of usage of common compound words and phrases, there can be significant variation for rarer combinations. Technical terms often show variation between disciplines. One study[308] found 15 different forms of the term *Epithelial cell*.

The formation of compound words in English is discussed in more detail elsewhere.

792 compound word

3.3.1 Common and rare word characteristics

Are there any differences in the characteristics of common and rare words?

An often-quoted study[803] investigated whether there were any differences in characteristics between high- and low-frequency English words (apart from their frequency of occurrence). The analysis suggested that differences existed; however, the analysis used four-letter words only, and words at the two frequency extremes. A later study by Frauenfelder, Baayen, Hellwig, and Schreuder[449] performed a more extensive analysis, for English and Dutch, using words containing between three and eight characters, with a range of frequencies.

Several of the Landauer et al. results (e.g., neighborhood density is higher for high-frequency words) were replicated for the case of four-letter English words. However, their findings did not hold across all word lengths. The results, for Dutch words, showed a weak but significant correlation between neighborhood density and word frequency. Although other differences were found (both for English and Dutch words), the authors were not able to find any significant word-frequency effects that applied across more than a few words lengths.

792 neighborhood identifier

A study by Howes and Solomon[599] found that the time taken to identify a word was approximately a linear function of the log of its relative frequency (the 75 most common words in the Thorndike–Lorge word counts were used).

3.3.2 Word order

Identifier names are sometimes built up from a sequence of words corresponding to a phrase or short sentence in a natural language familiar to the developer. Given a developer's natural language experience, the order of

these words is likely to be the one that has semantic meaning in that language.

A few natural languages permit arbitrary word order, but in most cases the order used will have a semantic significance in the language used. For instance, the English sentence "Tanya killed Masha" has a different meaning than "Masha killed Tanya", while the words in the equivalent Russian sentence "Tanja ubila Mašu" could appear in any of the six permutations of the three words and still be a grammatically valid sentence with the same meaning. (However, the order SVO is the most frequently used in Russian; other languages having free word order also tend to have a frequently used order.)

The three principle components of a natural language sentence are the *object*, *subject*, and *verb*. While the order in which these three components occur can vary within a single language, most languages have a preferred order (the frequency with which different orders occur within a particular language will depend on whether it uses a pragmatic word order— *topic-prominent* such as in Chinese and Japanese— or uses a grammatical word order — *subject-prominent* such as in English and Spanish). Much of the material in this subsection is derived from *Language Universals and Linguistic Typology* by Bernard Comrie.[265]

Table 792.7: Known number of languages commonly using a particular word order. Based on Comrie.[265]

Common order	Languages	Example
None	no figures	Sanskrit
SOV	180	Turkish "Hansan ököz-ü aldı" ⇒ "Hassan ox bought"
SVO	168	English "The farmer killed the duckling"
VSO	37	Welsh "Lladdodd y ddraig y dyn" ⇒ "killed the dragon the man"
VOS	12	Malagasy "Nahita ny mpianatra ny vehivavy" ⇒ "saw the student the woman"
OVS	5	Hixkaryana "Toto yahosi-ye kamara" ⇒ "man it-grabbed-him jaguar"
OSV	2	Apurinã none available

There are three other major word-order parameters, plus many minor ones. The following are the major ones:

- Within a sentence a noun may be modified by an adjective (becoming a *noun phrase*). The two possible word orderings are AN (English "the green table") and NA (French "le tapis vert" ⇒ "the carpet green"). Languages that use the order NA are more tolerant of exceptions to the rule than those using the order AN.
- A noun phrase may also contain a possessive (the technical term is *genitive*). The two possible word orderings are GN (Turkish "kadın-ın çavuğ-u" ⇒ "woman chicken-her") and NG (French "la plume de ma tante" ⇒ "the pen of my aunt"). English uses both forms of possessive (the Saxon genitive "the man's hat" and the Norman genitive "the roof of the house"). Although the Norman genitive is the more frequent, it is not possible to specify that it is the basic order.
- A language may contain prepositions, Pr (English "in the house"), or postpositions, Po (Turkish "adam için" ⇒ "man for the").

There are 24 possible combinations of object/subject/verb, noun/adjective, noun/genitive, and preposition/postposition that can occur in languages. However, empirical data on existent languages shows the following common patterns:

- VSO/Pr/NG/NA
- SVO/Pr/NG/NA
- SOV/Po/GN/AN
- SOV/Po/GN/NA

As well as using an order for different kinds of words, speakers also preferentially order the same kinds of words; for instance, the relative order in which adjectives occur.

word order 792
adjectives

3.4 Semantics

Identifier semantics

The hypothesis that all human languages have the same expressive power and are expressively complete in the sense that a proposition that can be expressed in one of them can be expressed in any of them is known as the *effability principle*. However, propositions that can be succinctly expressed in one language may require a great many words to be expressed in another language.

Cross-language research has shown that there are very few concepts (they mostly relate to the human condition) that might be claimed to be universal. For a discussion of semantic universals across languages see Wierzbicka[1461] and more recently von Fintel and Matthewson.[1428]

language affecting thought

The extent to which the language used by a person influences their thought processes has been hotly debated over the centuries; more recently researchers have started to investigate how thought processes influence language use (Lucy[871] provides a detailed history). The proposal that language does influence thought is commonly known as the *Sapir-Whorf* or *Whorfian* hypothesis. Some people hold what is known as the *strong language-based* view, believing that the language used does influence its speakers' conceptualization process. People holding the so-called *weak language-based* view believe that linguistic influences occur in some cases. The *language-as-strategy* view holds that language affects speakers performance by constraining what can be said succinctly with the set of available words (a speed/accuracy trade-off, approximating what needs to be communicated in a brief sentence rather than using a longer sentence to be more accurate).[607]

3.4.1 Metaphor

Metaphor

A data structure containing information about a politician's past record might include information about elections for which they have been a candidate. In the US politicians *run* for office, while in Spain and France they *walk*, and in Britain they *stand* for office. These are metaphors, and developers are likely to make use of them in the naming of identifiers (e.g., `ran_for`, `is_standing`).

Concepts involving time are often expressed using a spatial metaphor. These metaphors take two forms—one in which time is stationary and we move through it (e.g., "we're approaching the end of the year"); in the other case, we are stationary and time moves toward us (e.g., "the time for action has arrived").

A study by Boroditsky[139] investigated subject's selection of either the ego-moving or the time-moving frame of reference. Subjects first answered a questionnaire dealing with symmetrical objects moving to the left or to the right. The questions were intended to prime either an ego-moving or object-moving perspective. Subjects then read an ambiguous temporal sentence (e.g., "Next Wednesday's meeting has been moved forward two days"). The results found that 71.3 subjects responded in a prime-consistent manner. Of the subjects primed with the ego-moving frame, 73.3% thought the meeting was on Friday and 26.7% thought it was on Monday. Subjects primed with the object-moving frame showed the reverse bias (30.8% and 69.2%).

For a readable introduction to metaphors in everyday English see Lakoff and Johnson.[797]

3.4.2 Categories

Studies of color categorization provide a good example of the interaction between how peoples bodies work (in this case the eye), the category members supported by a language (in this case the different basic color terms), and human perception (see Hardin and Maffi[541] for an up-to-date discussion).

It was once thought that, across languages, color categories were arbitrary (i.e., the color terms used by different languages divided up the visible spectrum differently). In a now-classic study of 98 languages Berlin and Kay[115] isolated what they called the *basic color terms*. While the boundaries between color terms varied, the visual appearance of the basic color terms was very similar across languages (color matching has been found to be driven by physiological factors in the eye). They also found that the number and kind of basic color terms in languages followed a consistent pattern (see Figure 792.9).

A survey of empirical behavioral and linguistic uses of color term studies by Corbett and Davies[281] found (languages studied included English, Russian, Japanese, French, Hebrew, and Spanish) that:

- time taken to name a color was faster for the basic color terms;
- when asked to name a color, the basic color terms were usually listed first;

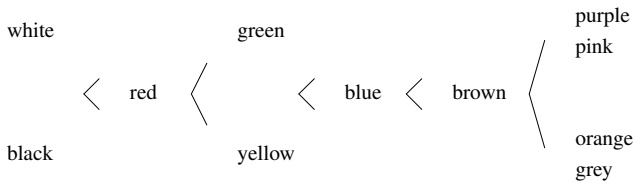


Figure 792.9: The original Berlin and Kay[115] language color hierarchy. The presence of any color term in a language, implies the existence, in that language, of all terms to its left. Papuan Dani has two terms (black and white), while Russian has eleven. (Russian may also be an exception in that it has two terms for blue.)

- the rank frequency of basic color terms in text matched that predicted by Berlin and Kay (more occurrences of the more basic color terms).

English is the *world language* of business (and software development) and a study of Japanese by Stanlaw[1282] found that English color loan words were supplanting the native ones (in an inverse order to the Berlin and Kay sequence).

3.5 English

words English

Languages have evolved complex processes for expressing subtle ideas. Although these processes may be obvious to native speakers, they tend to be unappreciated by inexperienced non-native speakers. This subsection uses English as an example for a discussion of some of the complexities of a human language. For a detailed discussion of English word formation, see Bauer[100] or Selkirk.[1213] Bauer[101] discusses international varieties of English, as used by native speakers. Pinker[1089] provides a readable discussion of word rules, taking as his subject regular and irregular verbs.

implicit 0 learning

Many of the points covered in the following subsections will be familiar to those developers who speak English as a native language (although in some cases this knowledge will be implicit). The intent is to show to these developers the complexities of the language constructs they take for granted. Knowledge of, and practice using, these complexities takes many years of practice. Native speakers acquire it *for free* while growing up, while many non-native speakers never acquire it.

English optimized for

Analysis of the properties of English words suggest that they are optimized for recognition, based on their spoken form, using their initial phonemes.[303, 1215, 1406]

phoneme 792

Technically, the terms *consonant* and *vowel* refer to spoken sounds— phonemes. In the written form of English individual letters are not unambiguously one or the other. However, the letters *a*, *e*, *i*, *o*, and *u* often represent vowel sounds. In some contexts *y* represents a vowel (e.g., *nylon*) and in some contexts *u* does not represent a vowel (e.g., *quick*).

3.5.1 Compound words

compound word

The largest group of compounds is formed using two nouns; noun+noun ⇒ stone wall, rainbow. (A study of technical terms in a medical database[308] found that 80% of unique multiword terms involved two nouns; see Costello[288] for a computational model.) Other compound forms include: verb+noun ⇒ *killjoy*, *spoilsport*; noun+verb ⇒ *homemade*, *rainfall*; adjective/adverb+noun ⇒ *quick-frozen*, *nearsighted* (see Costello[289] for a computational model); preposition+noun ⇒ *overload*, *underdog*; preposition+verb ⇒ *underestimate*, *overstep*; verb+particle ⇒ *makeup*, *breakdown*.

The creation and use of noun+noun compounds has been the subject of a number of studies. These studies have looked for restrictions that English speakers might place on permissible combinations (none found by Downing[368]) and the attributes associated with the individual words used to form the meaning of the new word.

A noun+noun combination has two parts in English. The first word acting as the modifier concept, while the second word is the head concept. Surveys[287, 1477] have found the main kinds of combinations that occur are:

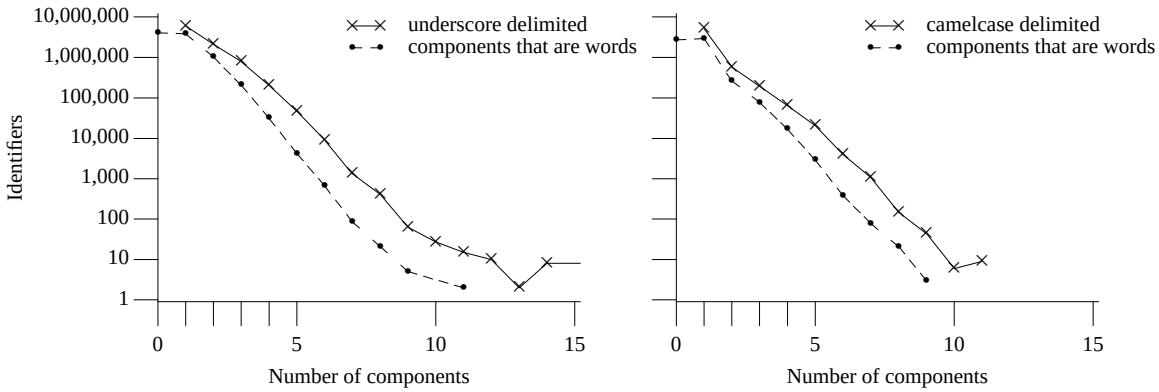


Figure 792.10: Number of identifiers containing a given number of *components*. In the left graph a component is defined as a character sequence delimited by one or more underscore characters, _, the start of the identifier, or its ending, e.g., the identifier `big_blk_proboscis` is considered to contain three components, one of which is a word. In the right graph a component is any sequence of lower-case letters, a sequence of two or more upper-case characters (i.e., the sequence is terminated by one or more digit characters or a letter having opposite case), or an upper-case character followed by a sequence of lower-case letters (this form of identifier might be said to be written in *camelCase*). For instance, the identifier `bigBlk4proboscis` is considered to contain three components, one of which is a word. A word is defined by the contents of the `ispell` 65,000 word list (this means, for instance, that the character sequence proboscis is not considered to be a word). Based on the visible form of the `.c` files.

- *conjunctive*, where concepts from both words are combined; for instance, *pet bird* is a bird that is also a pet. These have been found to occur in less than 10% of compounds.
- *property*, where a property is transferred from one of the concepts to the other; for instance, an *elephant fish* is a big fish. Like relational interpretations, these have been found to occur between 30% to 70% of the time. The frequency of property combinations has been found to increase if the concepts are similar to each other.[1476] For a study comparing the different theories of property combinations, see Costello.[290]
- *relational*, where a relation exists between two concepts; for instance, an *apartment dog* is a small dog that lives in city apartments. These have been found to be occur 30% to 70% of the time.

Words used to form an identifier might be a noun phrase rather than a compounded word. In this case word order differences may exist among dialects of English; for instance, British English uses *River Thames*, while American English uses *Hudson River*.

3.5.2 Indicating time

Tense is the linguistic term given to the way languages express time. In English tense is expressed using a verb and is divided into three zones: past, present, and future (see Table 792.8). Not all languages provide an explicit method of denoting the three tenses available in English; for instance, Japanese and Russian have no future tense.

Table 792.8: The 12 *tenses* of English (actually three tenses and four aspects). Adapted from Celce-Murcia.[213]

	Simple	Perfect	Progressive	Perfect progressive
Present	write/writes walk/walks	has/have written has/have walked	am/is/are writing am-is/are walking	has/have been writing has/have been walking
Past	wrote walked	had written had walked	was/were writing was/were walking	had been writing had been walking
Future	will write will walk	will have written will have walked	will be writing will be walking	will have been writing will have been walking

Time concepts that can be expressed in some other languages include distinctions between now versus not now and approximate versus not approximate.

3.5.3 Negation

English negation

The meaning of adjectives and adverbs can be inverted by adding a prefix (or sometimes a suffix). The following examples are from Celce-Murcia:[213]

```
happy ⇒ unhappy
appropriate ⇒ inappropriate
possible ⇒ impossible
logical ⇒ illogical
relevant ⇒ irrelevant
ordered ⇒ disordered
typical ⇒ atypical
life ⇒ lifeless
sense ⇒ nonsense
body ⇒ nobody
like ⇒ dislike
```

un- does not always indicate negativity; sometimes it indicates reversal (e.g., *unwrap*, *unfasten*). There are also words that do not follow this pattern (e.g., *inflammable* and *flammable* have the same meaning).

A noun can be negated by either adding *non-* or creating a non-phrase; for instance, *alternative* ⇒ *no alternative* and *sugar* ⇒ *sugar free*.

English also has two word forms that can be used to form a negation (i.e., not-negation and no-negation). Use of negation is much more common in speech that written text[121] (conversation is interactive and speakers have the opportunity to agree or disagree with each other, while written material usually represents the views of a single person). Studies of English grammar usage[121] have found that no-negation can be replaced with not-negation approximately 80% of the time (replacement in the other direction is possible in approximately 30% of cases), and that use of not-negation is several times more common than no-negation.

! 1103 operand type

The use of negation in C expressions is discussed elsewhere.

3.5.4 Articles

The English articles are: *definite* (e.g., *the* in "the book" would normally be taken to refer to a specific book), *indefinite* (e.g., *a/an* in "a book" does not refer to a specific book; the unstressed *some* is used for plural forms), and use of no article at all.

Most Asian and Slavic languages, as well as many African languages have no articles, they use article-like morphemes, or word order to specify the same information (e.g., the topic coming first to signal new information).

Experience shows that inexperienced users of English, whose native language does not have articles (e.g., Russian), have problems using the appropriate article. For instance, saying "have you book?" rather than"have you the book?" or "do you have a book?".

3.5.5 Adjective order

word order adjectives

In English adjectives precede the noun they modify. This is not always true in other languages. For instance, in French adjectives relating to age, size, and evaluation precede the noun, while adjectives referring to color or origin follow it:

```
une grande voiture jaune
    (big)   (car)  (yellow)
une vielle femme Italienne
    (old) (woman) (Italian)
```

Although it is rare for more than two adjectives to modify the same noun, the relative position of many of them has been found to have a consistent ordering. Svatko[1319] used responses from 30 subjects to deduce a probability for the relative ordering of certain kinds of adjectives (see Table 792.9).

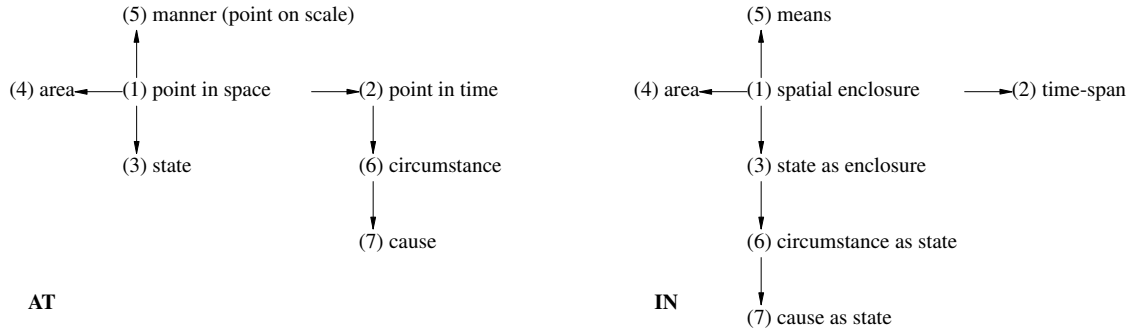


Figure 792.11: Examples, using "**at**" and "**in**" of extensions of prepositions from physical to mental space. Adapted from Dirven.[361]

Table 792.9: Probability of an adjective occurring at a particular position relative to other adjectives. Adapted from Celce-Murcia.[213]

determiner	option	size	shape	condition	age	color	origin	noun
	0.80	0.97	0.66	0.79	0.85	0.77	1.0	
an	ugly	big	round	chipped	old	blue	French	vase

3.5.6 Determine order in noun phrases

Within a noun phrase, determiners follow a general order; for instance:

```
pre core post
All our  many hopes . . .
core  post post
These next two weeks . . .
```

Table 792.10: Subcategories of determiners. Adapted from Celce-Murcia.[213]

Predeterminers	Core determiners	Post determiners
qualifiers: *all, both, half*, etc. fractions: *such a, what a*, etc. multipliers: *double, twice, three times*, etc.	articles: *a, an, the*, etc. possessives: *my, our*, etc. demonstratives: *this, that*, etc.	cardinal numbers: *one, two*, etc. ordinal numbers: *first, second*, etc. general ordinals: *next, last, another*, etc.
	quantifiers: *some, any, no, each, every, either, neither, enough*, etc.	quantifiers: *many, much, (a) few (a) little, several, more, less most, least*, etc. phrasal quantifiers: *a great deal, of, a lot of, a good number of*, etc.

Like adjective order, speakers of English as a second language often have problems using the appropriate determiner in the correct word order.

3.5.7 Prepositions

Prepositions are used to show role relationships. In some languages (e.g., Japanese) prepositions appear after the noun, in which case they are called *postpositions*. The same task is performed in some other languages (e.g., German, Russian) through the use of inflections. Prepositions

A single preposition can express a wide range of relationships (it said to be *polysemous*); for instance, the networks in Figure 792.11 highlight the relationships between the following phrases:

1. Point in space: "**at** the station", or spatial enclosure: "**in** the station"

2. Point in time: "**at** six o'clock", time-span: "**in** one hour"
3. State: "**at** work", or "**in** search of"
4. Area: "good **at** guessing", or "rich **in** coal"
5. Manner: "**at** full speed", or "**in** a loud voice"
6. Circumstance: "**at** these words (he left)", or "she nodded **in** agreement"
7. Cause: "laugh **at**", or "revel **in**"

Tyler and Evans[1382] give a detailed analysis of the range of meanings associated with spatial particles and provide a detailed analysis of the word *over*.[1381] Allen[15] gives a temporal algebra that can be used to describe intervals of time.

3.5.8 Spelling

English spelling

English spelling has many rules and exceptions. Proposals to regularize, or simplify, the spelling of English have more than 200 years of history.[1414] Experienced writers have learned to apply many of these rules and exceptions. For instance, the letter *e* is required after certain letters (*give*, *freeze*), or it may modify the pronunciation (*mat* vs. *mate*, *not* vs. *note*), or help to indicate that a word is not plural (*please*, *raise*), or it can indicate a French or Latin origin of a word (*-able*, *-age*).

A study by Venezky[1415] of the 20,000 most frequent words led him to create his *seven principles of English orthography*, covering the correspondence from writing to sounds.

1. *Variation is tolerated.* Words can have alternate spellings, either across dialects (American *honor* versus British *honour*) or within a single community (*judgment* vs. *judgement*).
2. *Letter distribution is capriciously limited.* Only a few of the letter combinations that are possible are permitted. For instance, doubling is prohibited for the letters a, i, h, v, z (there are a few exceptions like *skivvy*).
3. *Letters represent sounds and mark graphemic, phonological and morphemic features.*
4. *Etymology is honored.* That is, spelling relates to the history of English. For instance, a word that was borrowed early from French will have a */tS/* correspondence for *ch* (e.g., *chief*), while a word that was borrowed from French at a later period will have a */S/* correspondence (e.g., *chef*).
5. *Regularity is based on more than phonology.* For instance, there is the so-called *three letter rule*. All one- or two-letter words are function words[792.8] (e.g., *I*, *by*, *to*, *an*, *no* etc.), while content words have three or more letters (e.g., *eye*, *bye*, *two*, *Ann*, *know*).
6. *Visual identity of meaningful word parts takes precedence over letter-sound.* The claims that English spelling is illogical are often based on the idea that spelling should correspond to speech sounds. However, English spelling attempts to represent the underlying word forms (stripped of features attached to them by phonological rules), not represent sounds. English uses a lexical spelling system: one spelling, one morpheme, whatever the permutations of pronunciation; for instance, *cup*/*cupboard*, *critic*/*criticise*, *like*/*liked*/*likes*, *sign*/*signature*,

phonology 792

7. *English orthography facilitates word recognition* for the initiated speaker of the language rather than being a phonetic alphabet for the non-speaker.

In English the pronunciation of a particular sequence of letters can depend on their position in the word. For instance, the letter sequence *ghoti* could be pronounced as *fish* (*gh* as in *cough*, *o* as in *women*, and *ti* as in *nation*).[792.9]

When experienced English speakers are asked to spell spoken words, they are sensitive to the context in which the vowels occur,[1362] the position of the phoneme within the word,[1076] and other idiosyncratic factors.

[792.8] There are some rarely used words that are exceptions to this rule *e.g., *ox*, *ax* a US and old English spelling of *axe*) and specialist words such as *id*.

[792.9] George Bernard Shaw's original observation referred to the possible spelling of the sound */fish/*.

3.6 English as a second language

identifier English as second language

English is not only the *world language* of business, but also of software development. Information encoded in an identifier's spelling can only be extracted by readers if they are familiar with the English grammar, or English words, that it makes use of. Developers writing source that will include a non-native speaker of English readership might want to consider the benefits of restricting their usage of English to constructs likely to be familiar to this audience.

Even in those cases where developers appear to have near-native English-speaking ability there may be significant differences in their grammar usage.[280] Grammar plays a role in identifier spelling because, in English, words may have forms that reflect their grammatical role within a sentence. The learning of grammatical morphemes (e.g., *-ing*, *-s*) has been found to occur (in children and adults) in a predictable sequence.[503] The following list is based on Cook,[272] who also provides a good introduction to the linguistic factors involved in second-language teaching:

1. plural *-s* (e.g., "Girls go")
2. progressive *-ing* in present continuous form (e.g., "Girls going")
3. copula forms of *be* (e.g., "Girls are here")
4. auxiliary form of *be* (e.g., "Girls are going")
5. definite and indefinite articles— *the* and *a* (e.g., "The girls go" or "A girl go")
6. irregular past tense (i.e., verbs that do not have the form *-ed*)— (e.g., "The girls went")
7. third person *-s* (e.g., "The girl goes")
8. possessive *s* (e.g., "The girl's book")

The ordering of this sequence does not imply an order of difficulty in learning to use a construct. Studies[821] have found some significant variation in the ease with which learners acquire competence in the use of these constructs.

How many words does a speaker of English as a second language need to know? English contains (per *Webster's Third International Dictionary*, the largest dictionary not based on historical principles) around 54,000 word families (*excited*, *excites*, *exciting*, and *excitement* are all part of the word family having the headword *excite*). A variety of studies[995] have shown that a few thousand word families account for more than 90% of words encountered in newspapers and popular books. A variety of basic word lists,[995] based on frequency of occurrence in everyday situations, have been created for people learning English. All application domains have their own terminology and any *acceptable use* list of words will need to include these.

Non-native speaker's ability to extract information from identifiers created by native speakers may currently be the primary commercial developer language concern. However, the amount of source written (and therefore identifiers created) by non-native speakers continues to grow. The importance of native speakers, and speakers having a different first language to the original developer, to extract information from identifiers created by non-native speakers will grow as the volume of code increases. Handling different developer *interlanguages*[1320] (see Figure 792.12) is likely to be difficult.

There are lower-level reading issues associated with developers who have English as a second language, including (see Henser[562] for a survey of research on language-specific thoughts about bilinguals, and Carlo and Sylvester[199] for research on second-language reading):

792 neighborhood identifier

- A study by van Heuven, Dijkstra, and Grainger[1402] found that the orthographic neighborhood of both languages affected the performance of bilinguals.
- A study by Ziegler, Perry, Jacobs, and Braun[1512] investigated how identical words and nonwords (in some cases there were minor spelling differences) were read in English and German. They found that body neighborhood had a larger impact on naming time in English than German. The error rates were consistent across the various conditions tested at 1.8% and 3.7% for English and German, respectively.

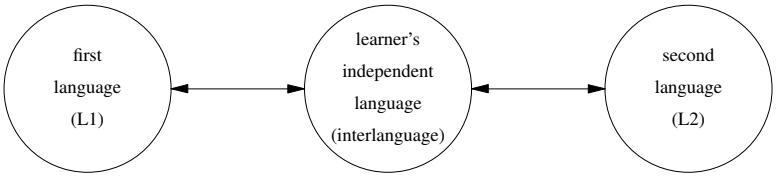


Figure 792.12: A learner's independent language— *interlanguage*. This language changes as learners go through the various stages of learning a new language. It represents the rules and structures invented by learners, which are influenced by what they already know, as they acquire knowledge and proficiency in a new language.

- A study by Malt and Sloman[891] asked second language users of English to name household objects (e.g., bottles, jars, plates, and bowls). They found that the names given by subjects did not fit the categorization patterns of native English speakers (subject performance was found to depend on the number of years of experience using English in a non-classroom setting). In the same way that the name applied to an object, by a native speaker, can depend on the context in which it occurs, there are cultural differences that affect the name given for an object.

context 792
naming affected by
naming 792
cultural differences

4 Memorability

identifier
memorability
memory 0
developer

An overview of human memory is given in Sentence 0. To summarize, short-term memory is primarily sound-based (some studies have found a semantic component), while long-term memory is semantics-based (meaning).[792.10]

This section discusses specific issues, those relating to identifiers, in more depth. In this coding guideline section memorability refers to the different ways developers recall information associated with an identifier in the source code. The kinds of information and their associations include:

- The spelling of one or more identifiers may need to be recalled. Spelling recall is usually indexed by semantic rather than letter associations. These associations may relate to information denoted by the identifier (e.g., the kind of information held in an object), its usage in the source code (e.g., a loop variable or the names of file scope objects modified by a call to a function), or some unique aspect of its declaration (e.g., a single parameter or label in a function definition).
- The information denoted by an identifier need may need to be recalled. This recall is usually indexed by the character sequence of an identifier's spelling.
- A previously seen identifier may need to be recognized as denoting the same entity when it is encountered again while reading source code.
- The locations in the source code that reference, or declare, an identifier may need to be recalled. This recall may be indexed by information on spelling or semantic associations.
- All the members of a list of identifiers may need to be recalled. The indexing information used for the recall and the kind of information required to be recalled varies with the identifier list and the context in which it is referenced. For instance, writing a **`switch`** statement whose controlling expression has an enumerated type requires knowledge of the names of the enumeration constants for that type, and use of designators in an initializer requires knowledge of the names of structure members. However, while a function invocation requires information on the expected arguments, that information is associated with the name of the function and the names of any parameters are rarely of interest.

designator 1648
. identifier

In most cases the required information is available in the source code (using identifiers in third-party libraries is one exception). However, readers would need to invest resources in locating it. The trade-offs people make when deciding whether to invest resources in locating information or to use information in their heads is discussed elsewhere.

cost/accuracy 0
trade-off

[792.10] Designers of IDEs ought to note that at least one study[1005] found that highlighting parts of a word did not produce any improvement in subject's recall performance.

The different kinds of developer interaction with identifiers places different demands on human memory resources. For instance, identifiers defined within functions are often referred to, by developers, as being *temporary* (a temporary object, temporary storage, or just a temporary). For a reader of the source the time interval over which they are *temporary* is the time needed to obtain the required information about the function being read. In the case of functions containing a few lines of code this might only be a few seconds, but in most cases it is likely to be more than a minute. During a day's work reading source code, a developer is likely to read many function definitions, each containing zero or more of these *temporary* identifiers, and some possibly sharing the same spelling.

792 identifier developer interaction

What is needed is the ability to be able to recall them while reading the body of the function that declares them and then to forget about them after moving on to the next function definition. Rather like walking out of a supermarket and recalling where the car is parked, but not being confused by memories for where it was parked on previous visits. Unfortunately, people do not have the ability to create and erase information in their memory at will.

While the brain mechanisms underlying human memory is a very active research area, it is not yet possible to give a definitive answer to any fundamental questions (although some form of semantic network for connecting related individual concepts is often used in modelling). Without a reliable model, it is not possible to describe and predict memory performance, and this section approaches the issue by trying to draw conclusions based on the various studies involving words and nonwords, that have been performed to date.

792 semantic networks

The following subsection discusses some of the studies that have been performed on human recall of different kinds of names (e.g., proper names and common names) and the various research projects aimed at finding out how people make the connection between a name and what it denotes. This is followed by three subsections that discuss the issues listed in the preceding three bullet points.

4.1 Learning about identifiers

It might be thought that developers would make an effort to remember the names of identifiers; for instance, by reading the locally declared identifiers when first starting to read the source of a function definition. However, your author's experience is that developers often read the executable statements first and only read declarations on an as-needed basis. Developers only need information on locally declared identifiers while reading the source of the function that contains them. The cost of rehearsing information about locally declared identifiers to improve recall performance is unlikely to be recouped. Whether looking up identifier information on an as-needed basis is the optimal cognitive cost-minimization technique is an open question and is not discussed further here.

What information do developers remember about identifiers? There is no research known to your author addressing this question directly. The two most common important memory lookup queries involving identifiers are their spelling and semantic associations. Before these two issues are discussed in the last two subsections of this section on memorability, there is a discussion on the results from various memory studies and studies of people's performance with proper names.

Information about identifiers is provided by all of the constructs in which they occur, including:

- A declaration (which may include an associated comment) provides information on the type, scope, and various other attributes. It is likely that the reader will want to recognize this declared identifier later and recall this information about it.
- An expression will reference identifiers that need to be recognized and information about them recalled. Being referenced by an expression is also potentially useful information to be remembered, along with the identity of other identifiers in the same expression.

The issue of how readers represent identifier information associated with declarations and expressions in memory is discussed elsewhere.

1348 declaration syntax
940 expressions

4.2 Cognitive studies

The power law of learning implies that the more often an identifier is encountered (e.g., by reading its name

0 power law of learning

memory 0
information
elaboration

or thinking about what it represents) the more likely it is to be correctly recalled later. Studies have also found that the amount of processing performed on to-be-remembered information can affect recall and recognition performance.

Human memory for pairs of items is not always symmetrical. For instance, a person who has learned to recall *B* when prompted with *B* might not recall *A* so readily when prompted with *B*. This issue is not discussed further here (see Kahana[704] for a discussion).

The performance of human memory can be depend on whether information has to be recalled or whether presented information has to be recognized. For instance, a person's spelling performance can depend on whether they are asked to recall or recognize the spelling of a word. A study by Sloboda[1255] asked subjects to choose which of two words was correctly spelled. The results showed (see Table 792.11) significantly more mistakes were made when the alternative was phonologically similar to the correct spelling (191 vs. 15); that is, the spelling looked sufficiently plausible.

Table 792.11: Example words and total number of all mistakes for particular spelling patterns (–*C*– denotes any consonant). Adapted from Sloboda.[1255]

Spelling pattern	similar phonologically	mistakes made	dissimilar phonologically	mistakes made
-ent -ant	clement clemant	46	convert convart	1
-ce -se	promice promise	9	polich polish	1
w- wh-	weight wheight	3	sapely shapely	1
-er -or	paster pastor	7	parret parrot	6
-le -el	hostle hostel	11	assits assist	1
-ayed -aid	sprayed spraid	18	slayer slair	0
-ea- -ee-	deamed deemed	24	dearth deerth	3
-CC- -C-	deppress depress	33	preessed pressed	0
-ancy -ency	currancy currency	27	corractly correctly	0
-al -el	rival rivel	13	livas lives	2

4.2.1 Recall

identifier
recall
initial letters 792
identifier
morphology 792
identifier

The initial letters of a word are a significant factor in several word related activities (it has been suggested[1326, 1328] that readers use information on the first few characters of a character sequence, researchers have yet to agree on whether orthographic, phonotactic, or syllabific boundaries are used, to build an internal representation that is then used to perform a search of their mental lexicon). The first study below describes their role in the *tip-of-the-tongue* phenomenon. The second gives an example of how context can affect recall performance. Recall is discussed in general, along with memory performance, elsewhere.

memory 0
developer

identifier
tip-of-the-tongue

- A study by Rubin[1184] investigated the so-called *tip-of-the-tongue* phenomenon. People in the tip-of-the-tongue state know that they know a word, but are unable to name it. Rubin gave subjects definitions of words and asked them to name the word (e.g., somebody who collects stamps, a *philatelist*). Those subjects who knew the word, but were unable to name it, were asked to write down any letters they thought they knew. They were also asked to write down the number of syllables in the word and any

similar-sounding words that came to mind. The results showed that letters recalled by subjects were often clusters at the start or the end of the word. The clusters tended to be morphemes (in some cases they were syllables). For instance, in the case of *philatelist* many subjects recalled either *phil* or *ist*.

identifier context cueing recall

- Context can also play an important role in cueing recall. A study by Barclay, Bransford, Franks, McCarrell, and Nitsch[93] investigated how cued recall varied with context. Subjects were divided into two groups and each group was shown a different list of sentences. For instance, the list of sentences seen by the members of one group might include— "The secretary put the paper clips in the envelope", while the other group would see a different sentence relating to secretaries and envelopes "The secretary licked the envelope". After seeing the sentences, subjects heard a list of cues and were asked to write down the noun from the list of sentences each cue reminded them of. Examples of cues included "Something that can hold small objects" and "Something with glue". It was predicted that cues matching the sentence context would produce better recall. For instance, the cue "Something that can hold small objects" is appropriate to paperclips (small objects) and envelopes (being used to hold something), but not directly to an envelope being licked (where the glue cue would have a greater contextual match). The results showed that subjects hearing cues matching the context recalled an average of 4.7 nouns, while subjects hearing cues not matching the context averaged 1.5 nouns.
- The visual similarity of words can affect serial recall performance. A study by Logie, Sala, Wynn, and Baddeley[865] showed subjects a list of words that was acoustically similar (to reduce the possibility of phonological information being used to distinguish them), but one set was visually similar (e.g., *FLY*, *PLY*, *CRY*, *DRY*) while the other set was visually distinct (e.g., *GUY*, *THAI*, *SIGH*, *LIE*). The results showed that the mean number of words recalled in the visually similar list was approximately 10% lower, across all serial positions, than for the visually dissimilar list.

4.2.2 Recognition

Recognition is the process of encountering something and remembering that it has been encountered before. Recognition is useful in that it enables previously acquired information to be reused. For instance, a reader may need to check that all the operands in an expression have a given type. If an identifier occurs more than once in the same expression and is recognized when encountered for the second time, the information recalled can remove the need to perform the check a second time.

Failing to recognize a previously seen identifier incurs the cost of obtaining the needed information again. Incorrectly recognizing an identifier can result in incorrect information being used, increasing the likelihood of a fault being introduced. The first study below describes one of the processes people use to work out if they have encountered a name before. The other two studies discuss how semantic context effects recognition performance. See Shiffrin and Steyvers[1228] for a recent model of word-recognition memory.

- A study by Brown, Lewis, and Monk[168] proposed that people use an estimate of a word's memorability as part of the process of deciding whether they had previously encountered it in a particular situation. For instance, names of famous people are likely to be more memorable than names of anonymous people. If presented with a list of famous names and a list of non-famous names, people are more likely to know whether a particular famous name was on the famous list than whether a non-famous name was on the non-famous list. The results of the study showed that in some cases name memorability did have an affect on subject's performance.
- A study by McDermott[913] asked subjects to memorize short lists of words. The words were chosen to be related to a nonpresented word (e.g., *thread*, *pin*, *eye*, *sewing*, *sharp*, and *thimble* are all related to *needle*). Subjects were then presented with words and asked to specify whether they were in the list they had been asked to memorize. The results showed that subjects recalled (incorrectly) the related word as being on the list more frequently than words that were on the list. The effect persisted when subjects were explicitly told not to guess, and a difference of 30 seconds or 2 days between list learning and testing did not change the observed pattern.

- A study by Buchanan, Brown, Cabeza, and Maitson[175] used a list of words that were either related to each other on a feature basis (e.g., *cat* and *fox* share the features: four legs, fur, tail, etc.) or by association (e.g., thread, pin, eye, sewing, thimble). The activation in a semantic network organized by features would be expected to spread to words that were related in the number of features they shared, while in an associated organization, the activation would spread to words that were associated with each other, or often occurred together, but did not necessarily share any features. The results showed that for an associated words list, subjects were much more likely to falsely recall a word being on the list, than when a feature-based words list was used.

identifier 792 phonetic symbolism

Recognition through the use of phonetic symbolism is discussed elsewhere.

4.2.3 The Ranschburg effect

Ranschburg effect

Some identifiers consist of a sequence of letters having no obvious pronunciation; for instance, `hrtmb`. In this case readers usually attempt to remember the individual letters of the sequence.

When the same letter occurs more than once in a letter sequence, a pattern of short-term memory behavior known as the *Ranschburg effect* occurs.[564] If occurrences of the same letter are separated by other letters, `hrtrb`, recall performance for the duplicate letter is reduced (compared to the situation where a nonduplicate letter appears at the same position in the sequence). If occurrences of the same letter occur together, `hrrtb`, recall performance for the duplicate letter is improved. This effect has also been found to occur for digits.[1459]

4.2.4 Remembering a list of identifiers

identifier learning a list of

In many contexts a sequence of identifiers occur in the visible source, and a reader processes them as a sequence. In some cases the identifiers in the sequence have a semantic association with each other and might be referred to as a list of identifiers. In other cases the only association connecting the identifiers is their proximity in the source.

```
1   typedef int ZIPS;
2
3   enum e_t {aa, bb, cc, dd};
4   struct s_t {
5               int mem_1;
6               long ee;
7              } xyz;
8
9   void f(int p_1, float foo)
10  {
11  ZIPS average;
12  int loc;
13  double bar;
14  /* ... */
15  bar=average+foo+xyz.ee-cc;
16  }
```

primacy 0 effect memory
recency 0 effect memory
identifier 792 confusability
identifier 792 semantic associations

A number of factors have been found to be significant when information on a sequence of identifiers needs to be recalled or remembered. The primacy and recency effects, confusability, and semantic issues are discussed elsewhere.

A study by Horowitz[594] illustrates how some of these factors affect subject's performance. Subjects were asked to learn a list of 12 trigrams. One list, known as *L4* was created using the four letters F, S, V, and X, while another list, known as *L12*, was created from 12 different consonants. Because there were only four letters to choose from, the trigrams in the first list often shared one or more letters with other trigrams in the list. The trigrams in the second list were chosen so that a particular pair of letters occurred in only one item.

A trial consisted of showing the subjects one of the lists of 12 trigrams. One group was then asked to write down as many as they could freely recall, while a second group had to arrange slips of paper (each containing a single, presented trigram) into the same order as the presentation. Subjects' performance was measured after each of 10 trials (each using a different order of the 12 trigrams).

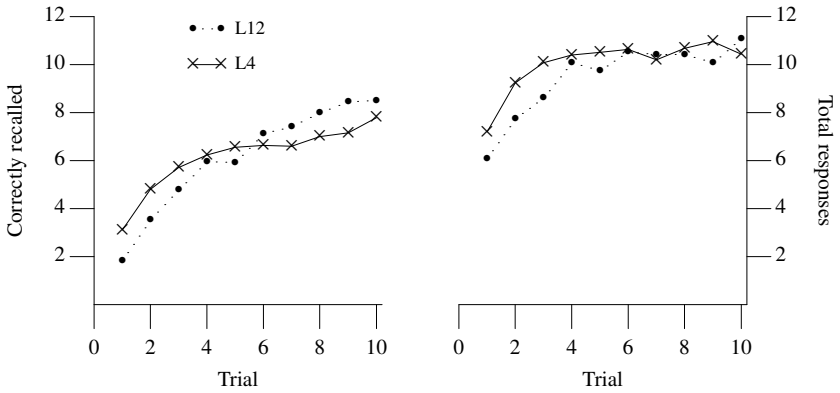


Figure 792.13: Mean correct recall scores and mean number of responses (correct and incorrect) for 10 trials. Adapted from Horowitz.[594]

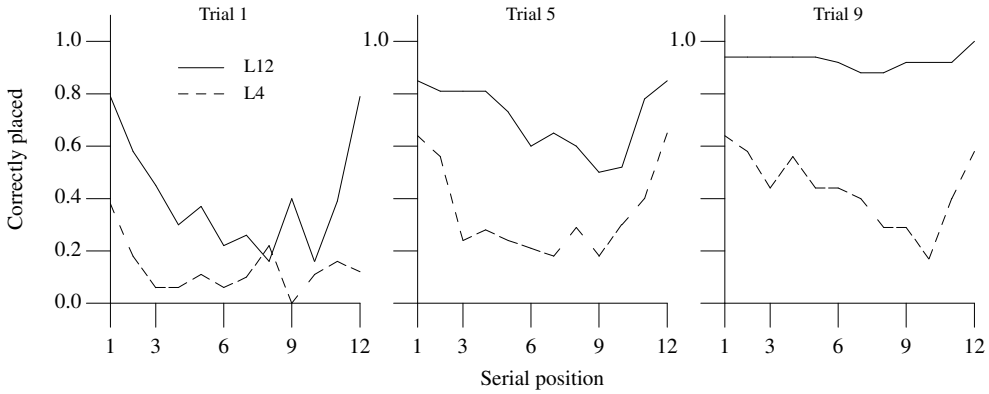


Figure 792.14: Percentage of correct orderings as a function of the trigram position within the list learned for three different trials. Adapted from Horowitz.[594]

The results for the free recall of trigrams (see Figure 792.13) show that initially subjects working with the L4 list performed best (the probability of randomly combining four letters to produce a correct three-letter trigram is 50%; this difference may be due to the much higher probability of being able to randomly pick a correct answer for the L4 list, compared to the L12 list). With practice (approximately six trials) recall performance of the group working with the L12 list exceeded that of the group using the L4 list.

The results for the ordering of trigrams (see Figure 792.14) show primacy and recency effects for both lists. The performance of subjects working with the L12 list is significantly better than those using the L4 list over all trials.

0 primacy effect memory
0 recency effect memory

A few studies have found that semantic information appears to be represented in short-term memory rather than simply being retrieved from long-term memory. A consequence of this representation is that semantic information associated with an identifier can affect recall performance. A study by Haarmann and Usher[530] showed subjects a list of six pairs of semantically related words and asked them to recall as many words as possible. In one list the semantically related words were adjacent to each other (e.g., "broader horizon king leader comfort uneasy kittens fluffy get purchase eating lunch"), while in the other they were separated by five unrelated words (e.g., "broader king comfort kittens get eating horizon leader uneasy fluffy purchase lunch"). The results showed that recall from the list of semantically adjacent words had an improved recency effect; that is, recall of words at the end of the list was better for semantically adjacent words than semantically separated words.

0 working memory information representation

0 recency effect memory

4.3 Proper names

A number of researchers have been investigating people's memory performance for proper names.[792.11] The results appear to show that the human brain does not handle different kinds of names in the same way. Studies of patients with aphasia (damage to the brain that causes problems with words) have found that different kinds of damage cause different kinds of problems. One study[506] found that some subjects had problems naming body parts but could name geographical locations, while others had problems naming geographical locations but could name body parts. Categories of names subject to these problems include animate versus inanimate objects, numbers versus letters, referential versus descriptive, and common versus proper names. Another study[504] found that brand names were handled differently in the brain than proper names.

Proper name recall performance has been found to decrease with age,[251] although in part this may be due to differences in mnemonic strategies (or lack of strategies).[165]

Many people experience greater difficulty remembering people's names than remembering other kinds of names. A number of possible reasons for this behavior have been proposed, including:

- Proper names are unique, and no alternatives are available. A study by Brédart[155] showed subjects the faces of people having two well-known names (e.g., Sean Connery alias James Bond and Peter Falk alias Columbo) and people having a well-known and a little-known name (e.g., Julia Roberts is not closely associated with any character's name she has played). Subjects were *blocked* (i.e., no correct answer given) on those faces having a single well-known name in 15.9% of cases, but were only *blocked* for 3.1% of faces having two well-known names (giving either name counted as a non-blocked response).

- The rate of learning of new common nouns slows as people reach adulthood. Adults learn technical names through further education and work experience, but learning new names in general usage is rare. However, people continue to learn new proper names throughout their lives. Being introduced to a *Mr. Dreaner* would not be considered unusual, but being told that a person was a *dreaner* might lead the listener to conclude they had misheard the job description. The range of plausible phonologies[157] is much greater for proper names than for common names. Having a set of known common names makes it easier to guess a particular name, given limited information about it (e.g., the first few letters).

A study by McWeeny, Young, Hay, and Ellis[924] asked subjects to learn to associate various names and professions with pictures of unfamiliar faces. Subjects were then presented with each picture in turn and asked to recall the associated name and profession. In those cases where only one association could be recalled there was a significant probability that it would be the profession rather than the name. In some cases the labels used could be either a name or a profession (e.g., Baker or Potter). Subjects asked to associate these ambiguous labels with a profession were more likely to recall them than those subjects asked to associate them with a name. This has been termed the *Baker–baker* paradox. The difficulty of recalling people's names is not directly related to the features of the name, such as its frequency of occurrence.

A number of other studies[1396] have confirmed that generally, recall of semantic properties associated with a person is faster than recall of that person's name. A study by Cohen[250] tested the hypothesis that the reason for this difference in performance was because peoples names are meaningless labels. There is no semantic association, in the subjects head, between the name and the person. In the first experiment subjects were asked to learn an association between a photograph, a name, an occupation, and either a meaningless nonword or a meaningful word (e.g., "Mr Collins; he is a teacher; he has a wesp", or "Mr Collins; he is a teacher; he has a boat").

[792.11] English, and most Indo-European languages, distinguish between two kinds of nouns. Names that denote individuals are called *proper names* (or *proper nouns*), while all other names (referring to classes of objects) are called *common names*.

Table 792.12: Mean number of each kind of information recalled in each condition (maximum score: 48). Adapted from Cohen.[250]

	Name	Occupation	Possession
Nonword	18.6	37.1	16.5
Word	23.6	37.0	30.4

The results (see Table 792.12) show that in the nonword case recall of both names and possessions was similar (the slightly greater recall rate for names could be caused by a frequency effect, the nonword names being names familiar to the subjects). When words were used for names and possessions, the relative recall rate changed dramatically. Cohen points out that information on real-world possessions has many semantic associations, which will trigger more connections in the subject's lexical network. A word used as a person's name is simply a label and is unlikely to have any semantic associations to an individual.

To what extent can parallels be drawn between different kinds of source code identifiers and different kinds of natural language names? For instance, are there similarities between the way developers treat the members of a structure and body parts, or between the way they think about labels and geographical locations? Identifiers do not always consist of a single word or non-word; they can form a phrase (e.g., `total_time`). Your author has not been able to find any studies looking at how human memory performance varies between words and phrases.

The two conclusions that can be drawn from studies about the recall of names is that richness of semantic associations can improve performance and that it is possible for different kinds of names to have different recall characteristics.

4.4 Word spelling

spelling

Spelling is the process of generating the sequence of characters that are generally accepted by native speakers of the language to represent the written form of a particular word. Software developers have extended this definition to include source code identifiers, whose names are commonly said to have a *spelling*.

Typing an identifier on a keyboard involves using memory to recall the sequence of letters required (or a rule to derive them) and motor skills to type the character sequence. This section provides a general discussion on the studies that have been made of spelling. The motor activities associated with typing are discussed elsewhere.

792 typing mistakes

Research into spelling has been carried out for a number of reasons, including learning about cognitive processes in the brain, trying to improve the teaching of children's reading and writing skills, and the creation of automated spelling correction programs . A lot of this research has used English speakers and words. It is possible that the models and theories applicable to users of a language that has a deep orthography may not apply to one that has a shallow orthography (e.g., Spanish[698]). Given that the spelling of identifiers is often more irregular than English, the availability of so much research on irregular spellings is perhaps an advantage.

Beware of heard,
a dreadful word,
That looks
like beard and
sounds like bird,
And dead:
it's said like
bed, not bead,
For Goodness'
sake, don't
call it deed!
Watch out
for meat and
great and threat,
They rhyme
with suite and
straight and debt.
– Anon

In some languages spelling is trivial. For instance, in Spanish if a person can say a word, they can spell it. Writers of other languages experience even more problems than in English. For instance, agglutinative languages build words by adding affixes to the root word. The effect of having such a large number of possible words (nouns have approximately 170 basic forms in Turkish and 2,000 in Finnish) on the characteristics of native speaker spelling mistakes is not known (automating the process of determining the root word and its affixes is a difficult problem in itself[1028]).

If people make spelling mistakes for words whose correct spelling they have seen countless times, it is certain that developers will make mistakes, based on the same reasons, when typing a character sequence they believe to be the spelling of an identifier. The following subsections discuss studies of spelling mistakes and some of the theories describing readers spelling performance. The aim is to find patterns in the mistakes made, which might be used to reduce the cost of such mistakes when typing identifier spellings. The fact that an identifier spelling may contain words, which may be misspelled for the same reasons as when they occur

792 agglutinative

in prose, is a special case.

4.4.1 Theories of spelling

How do people produce the spelling of a word? The two methods generally thought to be used are mapping from phonemes to graphemes (rule-based) and memory lookup (memory-based). The extent to which either of these methods is used seems to vary between people (those who primarily use the rule-based method are sometimes called *Phoenicians*, and those primarily using the memory-based method are called *Chinese*). A study by Kreiner and Gough[771] showed that good spellers made use of both methods.

Another possible method is that spelling is performed by analogy. A person uses a word for which the spelling is known which is phonemically or graphemically similar as a model, to produce the spelling of the unknown word.

4.4.2 Word spelling mistakes

The idea that there is a strong correlation between the kinds of spelling mistakes people make when writing prose and the kinds of spelling mistakes they make when writing C identifiers sounds appealing. Some broad conclusions about common prose spelling patterns can be drawn from studies of spelling mistakes (both written and typed). Kukich[784] provides a review of automatic spelling correction of text. However, the data on which these conclusions are based was obtained from disparate sources performing under very different conditions. Given the task and subject differences between these studies and developers writing code, any claims that these conclusions can be extended to developer performance in spelling identifiers needs to be treated with caution. For this reason the next subsection discusses the spelling mistake data, used in reaching these conclusions in some detail.

The following are some of the general conclusions drawn from the studies of spelling mistakes (a mistake is taken to be one of the operations insertion, deletion, substitution, or transposition):

- Between 69% to 94% of misspelled words contain a single instance of a mistake. The remaining misspelled words contain more than one instance.
- Between 1.4% to 15% of misspellings occurred on the first letter (this does not confirm the general belief that few mistakes occur in the first letter).
- Studies of typing performance have found strong keyboard adjacency effects (a letter adjacent to the one intended is typed). However, no spelling study has analyzed the effects of keyboard adjacency.
- There is insufficient data to analyze whether the number of mistakes made in a word is proportional to the number of letters in that word. (One study[771] found that the probability of a mistake being made increased with word length.)
- The pronunciation used for a word has a strong effect on the mistakes made. An incorrect spelling is often a homophone of the correct spelling.

Table 792.13: Breakdown of 52,963 spelling mistakes in 25 million typed words. Adapted from Pollock and Zamora.[1102]

Kind of Mistake	Percentage Mistakes
omission	34
insertion	27
substitution	19
transposition	12.5
more than one	7.5

Many of the spelling mistake data sets are derived from words people have chosen to use. However, people tend to limit the words they use in written prose to those they know how to spell.[966] Developers often have to use identifiers created by others. The characteristics of spelling mistakes for words chosen by other people

are of interest. An analysis by Mitton[946] looked at the differences in spelling mistakes made by 15 year-old children in written prose and a spelling test. The results show that, compared to mistakes in prose, mistakes in spelling test words contain more multiple mistakes in longer words (longer words are used less often in prose because people are less likely to be able to spell them).

A study by Adams and Adams[4] asked subjects to either spell a word (generation) or to select one of three possible spellings (recognition). Subjects also had to rate their confidence in the answer given. The results showed subjects were underconfident at low confidence levels and overconfident at high confidence levels, a commonly seen pattern. However, subjects' estimates of their own performance was more accurate on the recognition task.

0 overconfidence

Measuring spelling mistakes is not as straight-forward as it sounds. The mistakes made can depend on the subjects educational level (the educational level of many of the subjects at the time the data was obtained was lower than that typical of software developers, usually graduate-level), whether they or other people selected the words to be spelled, whether the mistakes were manually or automatically detected. Also, the English language accent, or dialect, spoken by a person has been found to affect word spelling performance for adults[1361] and children.[133, 346, 1291]

Like any other character, the space character can be mistyped. The difference between the space character and other characters is that it cannot occur within identifiers. An extra space character will cause a word to be split in two, which in C is very likely to cause a syntax violation. Omitting a space character is either harmless (the adjacent character is a punctuator) or will cause a syntax violation (in C). For these reasons, mistakes involving the space character are not discussed further here.

4.4.2.1 The spelling mistake studies

The spelling mistake studies and their findings are described next.

- Bourne[144] measured spelling mistakes in a bibliographic database. The results showed a misspelling rate that varied between 0.4% and 22.8% (mean 8.8%) for the index terms. The ratio of citations to misspelled terms varied between 0.01% and 0.63% (mean 0.27%).

- Kukich[783] analyzed spelling mistakes in the text messages sent over Bellcore's Telecommunications Network for the Deaf. Of the 10.5% of words automatically flagged as being in error, a manual check found that half were not in the dictionary used and the actual error rate was 5% to 6%

- Kundu and Chaudhuri[787] analyzed 15,162,317 of handwritten Bangla, a highly inflectional and phonetic language spoken in Indian, and 375,723 words of typed (mostly computer input) text. For the handwritten material, a mistake occurred every 122 words (0.82%) and in 53% of cases a word contained a single mistake (25% with two mistakes). The overall rate for typed text was 1.42%, and in 65% of cases a word contained a single mistake (11% with two mistakes).

- Mitton[946] looked at handwritten essays by 925 fifteen year old school children. The rate of mistakes of those classified as *poor* spellers was 63 per 1,000 words; it was 14 per 1,000 for *passable* spellers. The *poor* spellers formed approximately a third of the children and contributed more than half of the mistakes. An analysis of the Wing and Baddeley[1473] data found that few mistakes occurred near the start of a word.

- Pollock and Zamora[1102] (see Table 792.13) automatically flagged (using a dictionary of 40,000 words) mistakes in 25 million words from a scientific citations database. The overall rate of mistakes was 0.2%, and in 94% a word contained a single mistake (with 60% of mistakes being unique).

- Wing and Baddeley[1473] looked at the handwritten exam papers of 40 males applying to be undergraduates at Cambridge University (in engineering, mathematics, or natural sciences). The rate of mistakes was around 1.5% of words written. Of these, 52% were corrected by the subject after they had been made. Mistakes were less likely to be made at the beginning or end of words.

- Yannakoudakis and Fawthrop[1491] used information on 1,377 spelling error forms (obtained from published papers and checking written essays) to create what a list of "rules" for spelling errors (the mappings from correct one- or two-letter sequences to the incorrect one- or two-letter sequences were part of their data). They proposed that these spelling errors were based on phonological and letter sequence considerations; for instance, *BATH* may be pronounced *B-A-TH* in Northern England, but *B-AR-TH* or *B-OR-TH* in Southern England, potentially leading to different spelling errors. The letter sequence errors were mostly transpositions, letter doublings, and missed letters.

phonology 792

The words used in some analyses of spelling mistakes are based on words flagged by spell checking programs. These programs usually operate by checking the words in a document against a dictionary of correctly spelled words (letter trigram probabilities have also been used[1500]). One problem with the dictionary-based method is that some correctly spelled words will be flagged because they are not in the dictionary, and some incorrectly spelled words will not be flagged because the spelling used happens to match a word in the dictionary. As the number of words in the dictionary increases, the number of correctly spelled words that are flagged will decrease, but the number of unflagged spelling mistakes will also decrease. For instance, the rarely used word *beta* may be a misspelling of the more common *beat*, or its use may be intended. The word *beta* is unlikely to be in a small dictionary, but it will be in a large one. An analysis by Peterson[1081] found that the number of possible undetected spelling mistakes increases linearly with the size of the dictionary used (taking no account of word frequency). An analysis by Damerau and Mays[311] found that increasing the size of a spell checker's dictionary from 50,000 to 60,000 words eliminated the flagging of 3,240 correctly spelled words and caused 19 misspellings to be missed (in a 21,910,954 word sample).

4.4.3 Nonword spelling

identifier
nonword spelling

How do people spell nonwords (which may be dictionary words they are not familiar with)? For spoken languages a possible spelling might be based on the available sequence of sounds. Studies have found that, for English, people do not always spell a nonword spoken to them using the most common spelling pattern for the sound sequence heard. The choice of spelling is influenced by words they have heard recently. For instance, subjects who heard the word *sweet* just before the nonword */priːt/* tended to spell it as *preet*, while those who heard *treat* first tended to spell it as *preat*. Barry and Seymour[95] have proposed a model based on a set of probabilistic sound-to-spelling mappings and includes the influence of recently heard words.

4.4.4 Spelling in a second language

identifier
second language
spelling

A study by Cook[274] compared the spelling mistakes made by 375 overseas students at the University of Essex, against those made by the students in the Wing and Baddeley[1473] study. The results showed that students made fewer omissions (31.5% vs. 43.5%), but more substitutions (31.7% vs. 26.7%), transposition (3.1% vs. 1.4%), and other mistakes. Many of the mistakes were caused by close sound–letter correspondence (e.g., interchanging *a*, *e*, or *i*). There were many more different kinds of changes to consonants by overseas students compared to native speakers (38 vs. 21 different pairs).

A study by Okada[1033] investigated the kinds of English spelling mistakes made by native Japanese speakers. He proposed that the teaching and use of romaji (a method of representing spoken Japanese syllables using sequences of one or more letters from the English alphabet (only 19 to 21 of the 26 letters available are used, e.g., *c*, *l*, and *v* are not used)) was the root cause of particular kinds of spelling mistakes, i.e., subjects were using the romaji letter sequence/Japanese syllable sound association they were familiar with as an aid to spelling English words. The significant phonological differences between the spoken forms of Japanese and English can result in some spellings being dramatically incorrect. A later study[1034] looked at errors in the word-initial and word-final positions.

syllable 792

phonology 792

Your author does not know of any other study investigating the English spelling performance of native speakers of a language whose alphabet shares many letters with the English alphabet.

A study by Brown[167] compared the spelling abilities of native-English speakers with those for whom it was a second language (47 subjects whose first language varied and included Spanish, French, Japanese, Chinese, German, Hebrew, Arabic). The relative pattern of performance for high/low frequency words with

regular/irregular spellings (see Table 792.14) was the same for both native and non-native English speakers.

Table 792.14: Mean number of spelling mistakes for high/low frequency words with regular/irregular spellings. Adapted from Brown.[167]

	High Frequency Regular Spelling	Low Frequency Regular Spelling	High Frequency Irregular Spelling	Low Frequency Irregular Spelling
Native speaker	0.106	4.213	0.596	7.319
Second language	0.766	7.383	2.426	9.255
Example	cat, paper	fen, yak	of, one	tsetse, ghoul

4.5 Semantic associations

A semantic association occurs in the context of these coding guidelines when source code information causes other information to *pop* into a reader's mind. (It is believed[34] that people's recall of information from long-term memory is based on semantic associations.) Other semantic issues are discussed elsewhere.

792 identifier semantic associations
792 Identifier semantic confusability
792 semantic priming
792 Identifier semantic usability
792 semantic networks
770 reading kinds of

A particular identifier is likely to have several, out of a large number of possible, attributes associated with it. Which of these attributes a reader will be want to recall can depend on the purpose for reading the code and the kind of reference made in a usage of an identifier. The following are some of these attributes:

- *Identifiers in general.* What it denotes in the programs model of the application (e.g., `Node_Rec` might denote the type of a node record and `total_ticks` might denote a count of the number of clock ticks—its visibility), what other locations in the program's source reference the identifier (locations are usually the names of functions and relative locations in the function currently being read), associated identifiers not present in the source currently visible (e.g., a reference to a structure member may require other members of the same type to be considered), or who has control of its definition (e.g., is it under the reader's control, part of a third-party library, or other members of the project).
- *For an object.* The possible range of values that it might hold (e.g., `total_ticks` might be expected to always contain a positive value and a specification for the maximum possible value may exist), its type. (Some of the reasons for wanting to know this information include the range of possible values it can represent or whether it is a pointer type that needs dereferencing.)
- *For a function call.* A list of objects it references (for any call a subset of this list is likely to be needed, e.g., those objects also referenced in the function currently being read), or the value returned and any arguments required.

The following studies investigated the impact of semantic associations on recall of various kinds of information that might be applicable to developer interaction with identifiers.

- A study by McClelland, Rawles, and Sinclair[908] investigated the effects of search criteria on recall performance after a word-classification task. Subjects were given a booklet with pages containing categories and an associated list of words. For instance, the category *dangerous fish* and the words *shark*, *poison*, *trout*, and *paper*. They were asked to write down how many components (zero, one, or two) each word had in common with the category. After rating all category/word pairs, they then had to write down as many words from each of the lists as possible. The results showed that those words sharing two components with the category (e.g., *shark*) were most frequently recalled, those sharing one component with the category (e.g., *poison*, *trout*) were recalled less often, and those sharing no components (e.g., *paper*) were recalled least often. A second experiment measuring cued recall found the same pattern of performance. A study by Hanley and Morris[540] replicated these results.
- Several studies[869, 1182] have investigated readers incidental memory for the location of information in text printed on pages. The results show that for short documents (approximately 12 pages) subjects were able to recall, with reasonable accuracy, the approximate position on a page where information

occurred. These memories were incidental in that subjects were not warned before reading the material that they would be tested on location information. Recall performance was significantly lower when the text appeared on a written scroll (i.e., there were no pages).

5 Confusability

identifier confusability

For any pair of letter sequences (an identifier spelling) there is a finite probability that a reader will confuse one of them for the other. Some studies have attempted to measure confusability, while others have attempted to measure similarity. This section discusses the different ways readers have been found to treat two different character sequences as being the same. The discussion of individual issues takes the lead from the particular study being described in either using the term *confusability* or *similarity*. The studies described here involve carrying out activities such as searching the visual field of view, reading prose, recalling lists, and listening to spoken material. While all these activities are applicable to working with source code, the extent of their usage varies.

The following are the character sequence confusability, or similarity, factors considered to be important in this subsection:

- *Visual similarity.* Letter similarity, character sequence similarity, word shape (looks like)
- *Acoustic confusability.* Word sounds like, similar sounding word sequences
- *Semantic confusability.* Readers' existing knowledge of words, metaphors, categories

Metaphor 792

A reader of a C identifier may consider it to be a word, a pronounceable nonword, an unpronounceable nonword, or a sequence of any combination of these. This distinction is potentially important because a number of studies have shown that reader performance differs between words and nonwords. Unfortunately many of the published studies use words as their stimulus, so the data on readers' nonword performance is sparse.

word non-word effects 792

When categorizing a stimulus, people are more likely to ignore a feature than they are to add a missing feature. For instance, *Q* is confused with *O* more often than *O* is confused with *Q*. A study by Plauché, Delogu, and Ohala[1092] found asymmetries in subjects' confusion of consonants. For instance, while the spoken consonant */ki/* was sometimes interpreted by their subjects as */ti/*, the reverse did not occur (*/ki/* contains a spectral burst in the 3 to 4 KHz region that is not present in */ti/*).

distinguishing features 770

guideline recommendation enforceable 0

It is intended that these guideline recommendations be enforceable. This requires some method of measuring confusability. While there are no generally applicable measures of confusability, there is a generally used method of measuring the similarity of two sequences (be they letters, phonemes, syllables, or DNA nucleotides)— the Levenstein distance metric. The basic ideas behind this method of measuring similarity are discussed first, followed by a discussion of the various attributes, which might be sources of confusion.

5.1 Sequence comparison

Levenstein distance

The most common method used to measure the similarity of two sequences is to count the minimum number of operations needed to convert one sequence into the other. This metric is known as the *edit* or *Levenstein* distance. The allowed operations are usually insertion, deletion, and substitution. Some variants only allow insertion and deletion (substitution is effectively one of each), while others include transposition (swapping adjacent elements).

The Levenstein distance, based on letters, of *INDUSTRY* and *INTEREST* is six. One of the possible edit sequences, of length six, is:

```
INDUSTRY delete Y ⇒ INDUSTR
INDUSTR delete R ⇒ INDUST
INDUST substitute D by R ⇒ INRUST
```

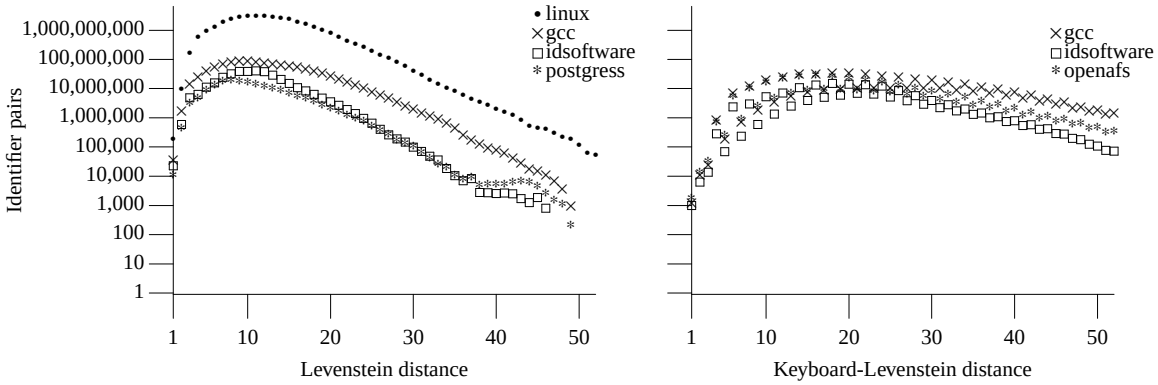


Figure 792.15: Number of identifiers having a given Levenstein distance from all other identifiers occurring in the visible form of the `.c` files of individual programs (i.e., identifiers in gcc were only compared against other identifiers in gcc). The *keyboard-levenstein* distance was calculated using a weight of 1 when comparing characters on immediately adjacent keyboard keys and a weight of 2 for all other cases (the result was normalized to allow comparison against unweighted Levenstein distance values).

```
INRUST substitute U by E ⇒ INREST
INREST insert T ⇒  INTREST
INTREST insert E ⇒ INTEREST
```

When all operations are assigned the same weight, the cost of calculating the Levenstein distance is proportional to the product of the lengths, $m \geq n$, of the two sequences. An $O(mn/\log n)$ algorithm is known,[900] but in practice only has lower cost for sequence lengths greater than 262,419. However, in its general form the Levenstein distance can use different weights for every operation. When operations can have different weights, the complexity of the minimization problem becomes $O(mn^2/\log n)$. Possible weighting factors include:

- A substitution may depend on the two items; for instance, characters appearing adjacent on a keyword are more likely to be substituted for each other than those not adjacent.
- The position of the character within the identifier; that is, a difference in the first character is much more likely to be visually noticed than a difference nearer the end of the identifier[106]. 770 word reading individual
- The visual similarity between adjacent characters; for instance, the Levenstein distance between the identifier `h1tk1` and the two identifiers `__t__` and `kltfh` (based on characters) is the same. However, the identifier `__t__` is much more different visually.

Computing an identifier's Levenstein distance, based on the characters it contains, to every other identifier in a program is potentially a computationally time-consuming operation. Possible techniques for reducing this overhead include the following:

- Reduce the number of identifiers against which the Levenstein distance has to be calculated
- Maintain a dictionary of precompute information for known identifiers. An algorithm by Bunke[180] looks up the nearest neighbor in a dictionary (which needs to hold identifiers converted to some internal form) in linear time, proportional to the length of one of the two strings to be matched (but storage usage is exponential in the length of the dictionary words).
- Map the identifiers into points in a d-dimensional space such that the distances between them is preserved (and less costly to calculate). Jin, Li, and Mehrota[669] describe an algorithm that uses this mapping idea to obtain the set of pairs of identifiers whose distance is less than or equal to k. While this algorithm runs in a time proportional to the total number of identifiers, it is an approximation that does not always find all identifier pairs meeting the distance criteria.

identifier 792 guideline significant characters

- The possibility of reducing the number of identifier's against which a particular identifier needs to be compared against is discussed elsewhere.

5.1.1 Language complications

Words often have an internal structure to them, or there are conventions for using multiple words. A similarity measure may need to take a reader's knowledge of internal structure and conventions into account. Three language complications are discussed next — an example of internal word structure (affixes), spelling letter pairs, and a common convention for joining words together (people's names), along with tools for handling such constructs.

agglutinative 792 languages

- A word sometimes includes a prefix or suffix. So-called *stemming* algorithms attempt to reduce a word to its common root form. Some algorithms are rule-based, while others are dictionary-based[774] (*creation* can be reduced to *create* but *station* cannot be reduced to *state*). Comparisons of the accuracy of the algorithms has produced mixed results, each having their own advantages.[460, 605] Languages differ in the extent to which they use suffixes to create words. English is at one end of the scale, supporting few suffixes. At the other end of the scale are Hebrew and Slovene (supporting more than 5,000 suffixes); Popovic and Willett[1103] found that use of stemming made a significant difference to the performance of an information lookup system.

morpheme 792

- There appears to be patterns to the incorrect characters used in misspelled words. Kernighan, Church, and Gale[728] used the Unix `spell` program to extract 14,742 *misspelled* words from a year's worth of AP wire text (44 million words). These words were used to build four tables of probabilities, using an iterative training process. For instance, $del[x, y]/chars[x, y]$ denotes the number of times the character y is deleted when it follows the character x (in misspelled words) divided by the number of times the character y follows the character x (in all words in the text). The other tables counted insertions, substitutions, and transpositions.

- People's names and source code identifiers often share the characteristic that blocks of characters are deleted or transposed. For instance, "John Smith" is sometimes written as "J. Smith" or "Smith John", and `total_widget` might be written as `tot_widget` or `widget_total`. Searching and matching problems based on different spellings of the same person's name occur in a variety of applications and a number of algorithms have been proposed, including, `LIKEIT`[1493] and `Matchsimile`.[996]

5.1.2 Contextual factors

The context in which a letter sequence is read can affect how it is interpreted (and possibly confused with another).

- *Reading sequences of words.* For instance, in the sentence "The child fed the dack at the pond" the nonword *dack* is likely to be read as the word *duck*. Sequences of identifiers separated by other tokens occur in code and are sometimes read sequentially. The extent to which identifier spellings will cause expectations about the spelling of other identifiers is not known.

- *Paging through source code* (e.g., when in an editor). Studies[1113] have found that when asked to view rapidly presented lists, subjects tend to mistakenly perceive a nonword as a word that looks like it (the error rate for perceiving words as other words that look like them is significantly smaller).

identifier 792 visual search

- *Searching.* The issues involved in visual search for identifiers are discussed elsewhere.

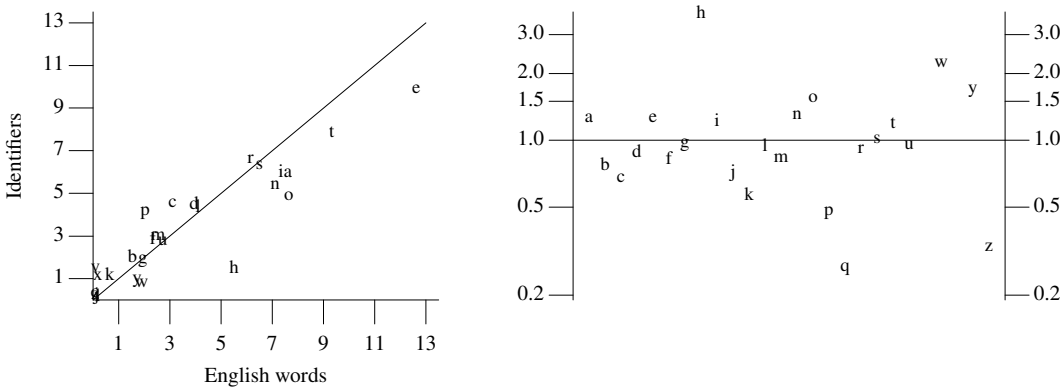


Figure 792.16: Occurrence of alphabetic letters in English text[1267] and identifier names (based on the visible form of the `.c` files; all letters mapped to lowercase). Left graph: the letter percentage occurrence as (x, y) coordinates; right graph: the ratio of dividing the English by the identifier letter frequency (i.e., letters above the line are more common in English text than in identifiers; two letters outside the range plotted are $v = 0.0588$ and $x = 0.165$).

5.2 Visual similarity

A large number of factors have been found to influence a reader's performance in visual recognition of character sequences (the special case of the character sequence representing a word is discussed elsewhere). This subsection discusses those visual factors that may cause one character sequence to be confused for another. Individual character similarity is discussed first, followed by character sequence similarity.

identifier
visual similarity
792 word
visual recognition

Readers have extensive experience in reading character sequences in at least one natural language. One consequence of this experience is that many character sequences are not visually treated as the sum of their characters. In some cases a character is made more visually prominent by the characters that surround it and in other cases it is visually hidden by these characters. Examples of these two cases include:

792 identifier
introduction

- For the first case, the *word superiority effect*. A study by Reicher[1152] showed subjects a single letter or a word for a brief period of time. They had been told that they would be required to identify the letter, or a specified letter (e.g., the third), from the word. The results showed that subjects' performance was better when the letter was contained within a word.
- For the second case, a study by Holbrook,[584] asked subjects to detect spelling errors in a 700-word story. The spelling errors were created by replacing a letter by either the letter with highest confusability, the letter with the lowest confusability, or one whose confusability was half way between the two extremes. After adjusting for letter frequency, word frequency, and perceived word similarity, the results showed a correlation between measurements of individual letter confusability and subjects' misspelling detection performance.

The following discussion assumes that all of the characters in a particular character sequence are displayed using the same font. Readers' performance has been found[1196] to be degraded when some of the characters in a word are displayed in different fonts. However, this issue is not discussed further here.

5.2.1 Single character similarity

The extent to which two characters have a similar visual appearance is affected by a number of factors, including the orthography of the language, the font used to display them, and the method of viewing them (print vs. screen). Examples of two extremes of similarity (based on commonly used fonts) are the characters `1` (one) and `l` (ell), which are very similar, and the characters `T` and `w` which are not visually similar.

character
visual similarity
792 orthography

In most cases the method used to view source code uses some form of screen. Reading from a printed listing is becoming rare. Even when a printed listing is used, it has usually been produced by a laser printer.

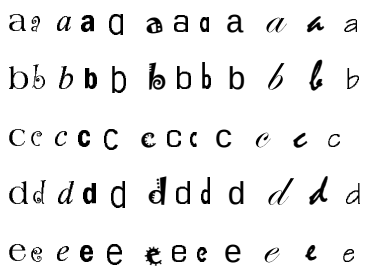


Figure 792.17: A number of different glyphs (different fonts are used) for various characters.

The character legibility issues caused by poorly maintained dot matrix or line printers are now almost a thing of the past and are not considered further here.

Before a character can be recognized, its shape has to be distinguished. The greater the visual similarity between two characters, the greater the probability that one of them will be mistakenly treated as the other. Studies of letter similarity measurement have a long history.[583, 1353] These studies have used letters only—there are no statistically significant published results that include digits or common mathematical symbols (as occur in C language source code). The raw data from these measurements is usually a two-dimensional confusion matrix, specifying the degree to which one letter has been estimated (by summing the responses over subjects) to be confusable with another one. In an attempt to isolate the different factors that contribute to this confusion matrix, a multidimensional similarity analysis is sometimes performed.

The following is a discussion of the major published studies (see Mueller et al[978] for a review). Kuennapas and Janson[782] asked subjects to judge the pairwise similarity of lowercase Swedish letters. A multidimensional similarity analysis on the results yielded nine factors: vertical linearity (e.g., *t*), roundness (e.g., *o*), parallel vertical linearity (e.g., *n*), vertical linearity with dot (e.g., *i*), roundness attached to vertical linearity (e.g., *q*), vertical linearity with crossness (e.g., *k*), roundness attached to a hook (e.g., *å*), angular open upward (e.g., *v*), zigzaggedness (e.g., *z*). Bouma[143] used Dutch subjects to produce confusion matrices for lowercase letters. The letters were viewed at distances of up to 6 meters and at angles of up to 10° from the center of the field of view. The results were found to depend on 16 different factors. Townsend[1360] used English subjects to produce confusion matrices for uppercase letters. The letters were briefly visible and in some cases included visual noise or low-light conditions (the aim was to achieve a 50% error rate).

A later study[1359] investigated the results from two individuals. Gilmore, Hersh, Caramazza, and Griffin[492] attempted to reduce the statistical uncertainty present in previous studies caused by small sample size. Each English uppercase letter was visually presented on a computer display, and analyzed by subjects a total of 1,200 times. A multidimensional similarity analysis on the published confusion matrices yielded five factors. Gervais, Harvey, and Roberts[482] attempted to fit their confusion matrix data (not published) for uppercase letters to models based on template overlap, geometric features, and spatial frequency content (Fourier transforms). They obtained a correlation of 0.7 between the data and the predictions of a model, based on spatial frequency content.

Boles and Clifford[134] produced a similarity matrix for upper- and lowercase letters viewed on a computer display (many previous studies had used letters typed on cards). A multidimensional similarity analysis on the results yielded three factors— lower- vs. uppercase, curved vs. straight lines, and acute angular vs. vertical.

Given that visual letter recognition is a learned process and that some letters occur much more frequently than others, it is possible that letter confusability is affected by frequency of occurrence. A study by Bouma[143] failed to find a frequency-of-occurrence factor in a letter's confusability with other letters.

5.2.2 Character sequence similarity

character sequence similarity

An important factor in computing character sequence similarity is a model of how people represent the

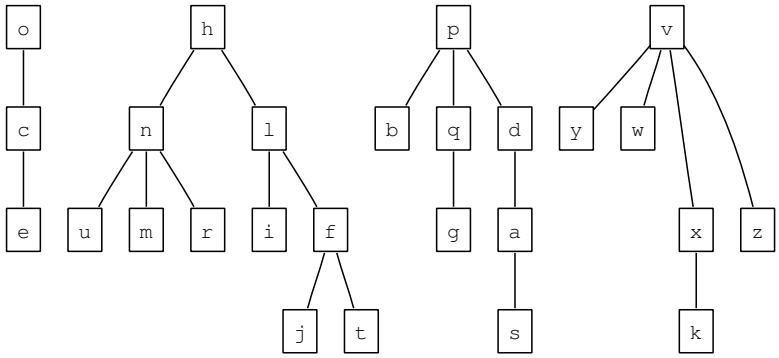


Figure 792.18: Similarity hierarchy for English letters. Adapted from *Lost reference*.[?]

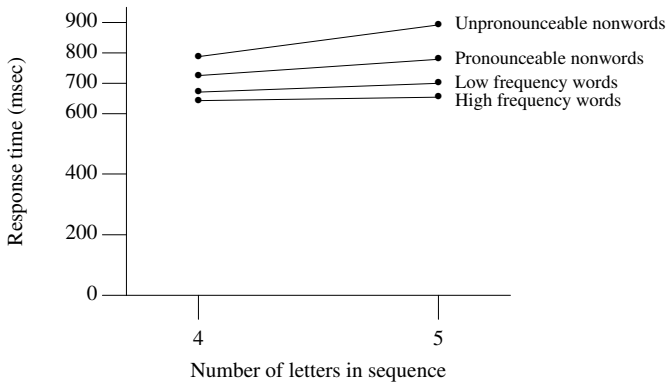


Figure 792.19: Response time to match two letter sequences as being identical. Adapted from Chambers and Foster.[214]

position of characters within a sequence. Proposed models include slot-coding (i.e., separate slots for each position-specific letter), localised context (i.e., the position of a letter with respect to its nearest neighbours), open-bigram encoding (e.g., *cl*, *ca*, *cm*, *la*, *lm*, and *am*).[328] Unfortunately researchers have only recently started to study this issue and there is not yet a consensus on the best model.

The following is a selection of studies that have investigated readers' performance with sets of character sequences that differ by one or more characters. The study by Andrews suggests that transposition may need to be included in a Levenstein distance calculation.

792 confusability
transposed-letter
792 Levenstein
distance

- A study by Chambers and Foster[214] measured response times in a simultaneous visual matching task using four types of letter sequences— high-frequency words, low-frequency words, pronounceable nonwords, unpronounceable nonwords. Subjects were simultaneously shown two letter sequences and had to specify whether they were the same or different. The results (see Figure 792.19) show the performance advantage for matching words and pronounceable nonwords. Measurements were also made when the letter sequences differed at different positions within the sequence; Table 792.15 shows the response times for various differences.

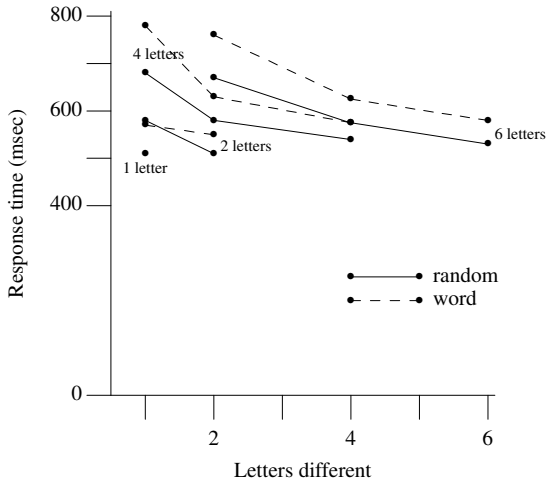


Figure 792.20: Time taken (in milliseconds) to match a pair of letter sequences as being identical— for different number of letters in the sequence and number of positions in the sequence containing a nonmatching letter. Adapted from Eichelman.[381]

Table 792.15: Response time (in milliseconds) to fail to match two letter sequences. Right column is average response time to match identical letter sequences. Columns are ordered by which letter differed between letter sequences. Adapted from Chambers and Foster.[214]

	All Letters	First Letter	Third Letter	Fifth Letter	Same Response
Words	677	748	815	851	747
Pronounceable nonwords	673	727	844	886	873
Unpronounceable nonwords	686	791	1,007	1,041	1,007

Chambers and Foster explained the results in terms of three levels of letter sequence identification: the word level, the letter cluster level, and the letter level. The higher the level at which operations are performed, the fewer are required for a fixed number of letters. The increasing response time as the differing letter moves further to the right of the word suggests a left-to-right order of processing. However, performance when all letters differ is much better than when only the first letter differs. This behavior would not occur if a strictly left-to-right comparison was performed and suggests some *whole word* processing occurs— also see Henderson.[559]

- A study by Eichelman[381] measured response times in a simultaneous visual matching task where letter sequences (either words or randomly selected) varied in the number letters that differed or in case (lower vs. upper). The results (see Figure 792.20) show how response time increases with the number of letters in a sequence and decreases as the number of letters that are different increases.

  In a second simultaneous visual matching task some letter sequences were all in uppercase, while others were all in lowercase. Subjects were told to compare sequences, ignoring letter case. The results showed that when the case used in the two letter sequences differed, the time taken to match them as being identical increased for both words and random sequences. This pattern of response is consistent with subjects performing a visual match rather than recognizing and comparing whole words. For this behavior, a change of case would not be expected to affect performance when matching words.

- Neighborhood frequency effects. A study by Grainger, O'Regan, Jacobs, and Segui[512] found an interaction (response time and error rate) between the initial eye fixation point within a word and the position of the letter differentiating that word from another (this behavior is only possible if readers have a preexisting knowledge of possible spellings they are likely to encounter, which they will have for words).

word 770
reading individual

Mr. Chips 770

- Perceptual interaction between two adjacent, briefly presented, words (as might occur for two operands in an expression when paging through source in an editor). For instance, subjects have reported the adjacent words *line* and *lace* as being *lane* or *lice*.[1399] 792 illusory conjunctions

- Transposed-letter confusability. A study by Andrews[37] found that so-called *transposed letter pair* words (e.g., *salt-slat*) affected subjects' performance in some cases (low-frequency words; non-words were not affected). A model of internal human word representation based on letters and their approximate position within a word was discussed. confusability transposed-letter

- A study by Ziegler, Rey, and Jacobs[1513] found that speed of recognition of words whose letters were not readily legible was proportional to the log of their frequency of occurrence and approximately linear on what they defined as letter confusability. The error rate for correctly identifying words was proportional to the number of orthographic neighbors of a word, and a measure of its orthographic redundancy.

5.2.2.1 Word shape

The term *whole word shape* refers to the complete visual appearance of the sequence of letters used to form a word. Some letters have features that stand out above (ascenders— *f* and *t*) and below (descenders — *q* and *p*) the other letters, or are written in uppercase. Source code identifiers may also have another shape-defining character— the underscore _. Words consisting of only uppercase letters are generally considered to have the same shape. The extent to which *whole word shape* affects visual word-recognition performance is still being debated (see Perea and Rosa[1074] for a recent discussion). In the following two studies a whole word shape effect is found in one case and not in the other. word shape

- A study by Monk and Hulme[958] asked subjects to quickly read a paragraph. As a subsidiary task they were asked to circle spelling errors in the text. The spelling errors were created by deleting or substituting a letter from some words (in some cases changing the shape of the word). The results (see Table 792.16) showed that when lowercase letters were used, a greater number of misspelled words were circled when the deletion also changed the shape of the word. When a mixture of letter cases (50% lowercase/uppercase) was used, differences in word shape were not sufficient to affect misspelling detection rates (this study has been replicated and extended[556]).

Table 792.16: Proportion of spelling errors detected (after arcsin transform was applied to the results). Adapted from Monk and Hulme.[958]

	Same Lowercase Word Shape	Different Lowercase Word Shape	Same Mixedcase Word Shape	Different Mixedcase Word Shape
Letter deleted	0.554	0.615	0.529	0.517
Letter substituted	0.759	0.818	0.678	0.680

- A study by Paap, Newsome, and Noel[1038] modified an earlier study by Haber and Schindler,[531] which had found a word shape effect. Subjects were asked to read a passage at their normal reading speed, circling any misspelled words as they read. They were told that they would be tested for comprehension at the end. Four kinds of misspellings were created. For instance, the letter *h* in *thought* was replaced by: (1) *b* (maintain shape and confusable letter), (2) *d* (maintain shape and distinct letter), (3) *n* (alter shape and confusable letter), or (4) *m* (alter shape and distinct letter). The results (see Figure 792.21) showed that many more misspellings went undetected when the letters were confusable (irrespective of word shape) than when they were distinctive (irrespective of word shape).

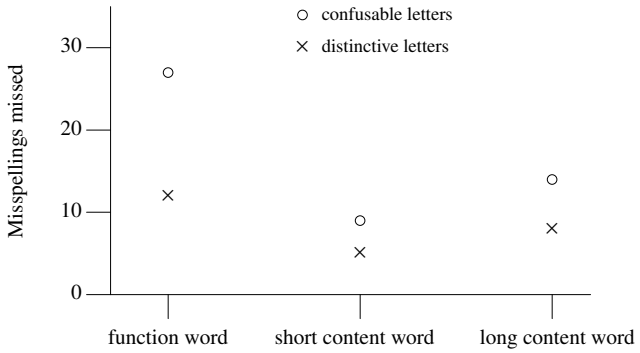


Figure 792.21: Percentage of misspellings not detected for various kinds of word. Adapted from Paap, Newsome, and Noel.[1038]

5.3 Acoustic confusability

acoustic confusability
standard 1 specifies form and interpretation
grapheme 792
phoneme 792
word 792 pronounceability

C source code is not intended to represent a spoken language. However, it is always possible to convert an identifier's character sequence to some spoken form. This spoken form may be a list of the individual characters or readers may map one or more characters (graphemes) to sounds (phonemes). This subsection does not concern itself with the pronunciation used (the issue of possible pronunciations a developer might use is discussed elsewhere), but discusses the various possibilities for confusion between the spoken form of different identifiers.

The studies discussed in this subsection used either native British or American speakers of English. It is recognized that the pattern of sounds used by different languages varies. The extent to which the results, relating to sound similarity, obtained by the studies discussed here are applicable to speakers of other languages is not known. While many of the studies on acoustic similarity use spoken material as input to subjects, not written material, most of the studies described here used written material.

A number of studies have found what has become known as the *dissimilar immunity effect*.[75] The effect is that the recall of phonologically dissimilar items on a list is not affected by the presence of similar items (i.e., recall is the same as if the items appeared in a completely dissimilar list). A recent study[409] found evidence that similar items enhanced memory for the order of dissimilar items between two similar items.

phonology 792

The first subsection discusses the issue of measuring the degree to which the pronunciations of two identifiers sound alike. This is followed by subsections discussing studies of letter acoustic confusion and memory performance for lists of words that sound alike.

5.3.1 Studies of acoustic confusion

A number of studies have investigated the acoustic confusion of letters and digits. Two of these (both using British subjects) published a confusion matrix. In the study by Conrad[268] (whose paper only contained data on a subset of the letters— see Morgan[963] for data on all letters), subjects listened to a recording of spoken letters into which some noise had been added and were asked to write down the letters they heard. The study by Hull[604] included letters and digits.

In a study by Morgan, Chambers, and Morton[964] subjects listened to a list of digits (spoken by either an American or Australian female or an English male) and after hearing the list were asked to write down the digits. A total of 558 subjects listened to and recalled 48,402 digit lists. Confusion matrices for the different speakers and the recognition and memory tasks were published.

What are the factors that affect the confusability of two letters? Both Morgan[963] and Shaw[1221] performed multidimensional scaling analysis[792.12] on the Conrad and Hull data. Morgan presented an analysis using $n = 3$, but suggested that larger values may be applicable. Shaw treated the letters phonologically, using

phonology 792

[792.12] Multidimensional scaling is a technique that attempts to place each result value in an n-dimensional space (various heuristics are used to select n, with each dimension being interpreted as a factor contributing to the final value of the result obtained).

$n = 5$ isolated phonological factors for three of the dimensions (open vs. closed vowels, vowel+consonant sound vs. consonant+vowel sound, and voiced vs. unvoiced consonants). There is significant correlation between the ordering of data points in one of the dimensions (open vs. closed vowels) and the first formant frequency of English vowels (see Cruttenden, tables 3, 4, and 5[301]). This correlation was also found by Morgan, Chambers, and Morton[964] in their study of digits.

While these results offer a possible explanation for the factors affecting letter confusion and data points from which calculations can be made, it is important to bare in mind that the pronunciation of letters will depend on a person's accent. Also, as the measurements in Cruttenden[301] show, the relative frequencies of formants (assuming this is a factor in confusability) differ between males and females.

Most identifiers contain more than one letter. A study by Vitz and Winkler[1426] found that for words the phonemic distance (a measure of the difference between two phoneme sequences) between the two words is a major factor in predicting whether people judge them to have "similarity of sound". Other factors include rhyme, alliteration, and stress.

792 word stress

A study by Bailey and Hahn[84] compared two of the main approaches to measuring phonological similarity (for the case of single syllable words), one empirically derived from measurements of confusability and the other based on a theoretical analysis of phonological features. They found that empirical confusability measurements did not provide any advantages over the theoretically based analysis.

5.3.1.1 Measuring sounds like

0 coding guidelines introduction

One of the design aims of guideline recommendations is to support automatic enforcement. In order to measure the acoustic similarity of two identifiers, it is necessary to be able to accurately convert their spellings to sound (or an accurate representation of their sound, e.g., a phonetic transcription; the issue of how people convert character sequences to sounds is discussed elsewhere) and to have a method of measuring the similarity of these two sounds. There are a number of issues that a measurement of sounds-like needs to address, including:

792 phoneme
792 characters mapping to sound

You can lead a horse to water, but a pencil must be lead. Wind the clock when the wind blows.

792 characters mapping to sound

- Developers are well aware that source code identifiers may not contain natural language words. It is likely that nonwords in identifiers will be pronounced using the grapheme-to-phoneme route (see Figure 792.3). For instance, the pronunciation of the nonword `gint` rhyming with the word `hint` rather than `pint` (of bear). A study by Monsell et al.[960] (and others) shows that readers have some control over the degree to which one naming route is preferred to another based on their expectations of the likely status of character sequences they will encounter. This suggests that a developer who expects identifiers to be nonwords may be more likely to pronounce identifiers using the grapheme-to-phoneme route. This route may, or may not, result in the generally accepted pronunciation for the word being used.
- Some identifiers contain more than one word. Readers recognize a word as a unit of sound. As such, two identifiers containing the same two words would have the same sound if those words occurred in the same order. For instance, `count_num` and `num_count` could be said to have a Levenstein distance of one based on the word as the edit unit.

A number of studies have attempted to automatically measure the sound similarity of two words (Kessler[729] provides an overview of phonetic comparison algorithms and the applications that use them), the following are some of them:

- Frisch[454] describes a method of computing a similarity metric for phonological segments (this is used to calculate a relative degree of confusability of two segments, which is then compared against English speech error data).

792 syllable

- Mueller, Seymour, Kieras and Meyer[977] defined a phonological similarity metric between two words based on a multidimensional vector of psychologically relevant (they claimed) aspects of dissimilarity. The dimensions along which the metric was calculated included the extent to which the words rhyme, the similarity of their stress patterns, and the degree to which their syllable onsets match.

792 phonology
792 syllable

- Lutz and Greene[879] describe a system that attempts to predict probable pronunciations of personal names based on language-specific orthographic rule sets. This information was used to automatically measure the phonological similarity between a name and potential matches in a database.

5.3.2 Letter sequences

Learning to read teaches people a method for mapping certain sequences of letters to sounds. Many adults have had so much practice performing this mapping task that it has become so automatic for very many sequences that the sound is heard rather than the letters seen.

A study by Lambert, Donderi, and Senders[800] asked each subject to classify 70 drug names, based on the individuals own criteria, by similarity. The results showed little correlation between these subjects' classification and a similarity measure based on counting letter trigrams common to each pair of drug names.

A study by Conrad[269] visually presented subjects with six consonants, one at a time. After seeing all the letters, subjects had to write down the recalled letters in order. An analysis of errors involving letter pairs (see Table 792.17) found that acoustic similarity (AS) was a significant factor.

Table 792.17: Classification of recall errors for acoustically similar (AS), acoustically dissimilar (AD) pairs of letters. *Semi-transpose* refers to the case where, for instance, *PB* is presented and *BV* is recalled (where *V* does not appear in the list). *Other* refers to the case where pairs are both replaced by completely different letters. Adapted from Conrad.[269]

Number Intervening Letters	Transpose (AS)	Semi-transpose (AS)	Other (AS)	Transpose (AD)	Semi-transpose (AD)	Other (AD)	Total
0	797	446	130	157	252	207	1,989
1	140	112	34	13	33	76	408
2	31	23	16	2	18	56	146
3	12	20	12	1	5	23	73
4	0	4	1	0	2	7	14
Total	890	605	193	173	310	369	2,630

5.3.3 Word sequences

As the following studies show, people recall fewer words from a list containing similar sounding words than dissimilar sounding words. The feature-based model of immediate memory[991] explains the loss of information from short-term memory in terms of interference between the items being remembered rather than their memory traces decaying over time. This model predicts that similar sounding words will interfere with each other more than dissimilar sounding words (it also accounts for the recency effect and temporal grouping). A subsequent enhancement to the model[1000] enabled it to account for the word-length effect (memory performance is worse for items that take longer to pronounce).

- A study by Baddeley[73] dramatically showed the effects of acoustic similarity on recall performance from short-term memory. Subjects were visually presented with either a list of acoustically similar words (e.g., *man*, *cab*, *can*, *cad*, *cap*, *mad*, *map*, etc.), or a list of words that were not acoustically similar (e.g., *few*, *pit*, *cow*, *pen*, *bar*, *hot*, *bun*, etc.). After a delay of zero, four, or eight seconds, during which they had to write down digits that were slowly read to them, they had to write down the presented word list. The results (see Figure 792.22) show the significant impact acoustic similarity can have on recall performance.
- A study by Daneman and Stainton[315] asked subjects to carefully read a passage (subjects were given a comprehension test) and proofread it for spelling errors. The spelling errors were either homophones (e.g., *meet* replaced by *meat*) or not (e.g., *meet* replaced by *meek*). Homophone misspellings were less likely to be detected than the non-homophone ones. A study by Van Orden[1403] asked subjects to make a semantic category decision on a visually presented word. For instance, they were told "answer yes/no if the word is a flower" and presented with either *ROSE*, *ROWS*, or *ROBS*. The results showed that subjects were significantly more likely to answer yes to words that were homophones of words that were members of the category.

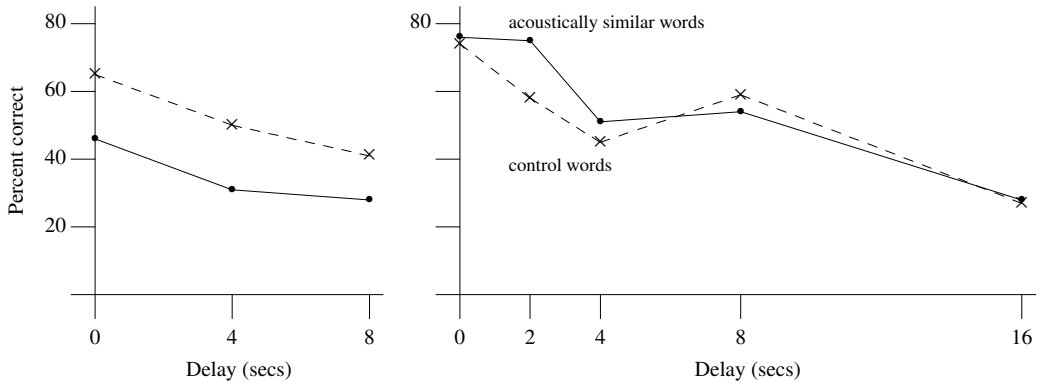


Figure 792.22: Rate of forgetting of visually presented lists of four words containing the same (solid line) or different vowels (dashed line); left graph. Rate for two lists, one containing three acoustically similar words (solid line) and the other five control words (dashed line); right graph. Adapted from Baddeley.[73]

- People do not always say the word they intended to speak. Two well-known kinds of mistake are: (1) *malapropisms*, using a word that has a similar sound but a different meaning ("the geometry of contagious countries"), and (2) *spoonerisms*, the transposition of syllables between two words (*blushing crow* instead of *crushing blow*). A study by Fay and Cutler[413] located 183 malapropisms in a collection of more than 2,000 speech errors. They found that in 99% of cases the target and erroneous word were in the same grammatical category, in 87% of cases they contained the same number of syllables, and in 98% of cases they shared the same stress pattern. 792 Spoonerisms

- *Silent* letters. Early studies[283] suggested that silent letters (i.e., not pronounced) in a word were more likely to be go unnoticed (i.e., their misspelling not detected in a proofreading task) than letters that were pronounced. Later studies[369] showed that the greater number of search errors occurred for high-frequency function words (e.g., *the*), not for content words. The search errors were occurring because of the high-frequency and semantic role played by the word, not because letters were either silent or pronounced.

- A study by Coltheart[261] found that it did not matter whether the same, visually presented, words were used on each memory trial, or different words were used every time; recall of similar sounding words was significantly lower (62–69% vs. 83–95% when recall in the correct order was required, and 77% vs. 85–96% when order-independent recall was required).

5.4 Semantic confusability

Identifier semantic confusability

Developers often intend identifier names to convey semantic information about what an identifier represents. Because of the small number of characters that are usually used, the amount of semantic information that can be explicitly specified in the name is severely limited. For this reason developers make assumptions (usually implicitly) about knowledge they share with subsequent readers. Semantic confusion occurs when a reader of the source does not meet the shared knowledge assumptions made by the creator of the identifier name.

This coding guideline section does not make any recommendations relating to semantic confusability. Given the extent to which people implicitly treat their own culture as *natural*, it is difficult to see how even detailed code reviews can reliably be expected to highlight culturally specific identifier naming assumptions made by project team members. However, the following subsections attempt to give a flavor of some of the possible confusions that might occur; the issues covered include natural language, metaphor, and category formation. 0 code reviews 792 Metaphor

This subsection primarily covers the issue from the perspective of a trying to minimise the confusion a developer may have over the semantics that might be associated with an identifier. It is possible for the

semantics associated with an identifier to be clear to a reader, but for a mistake made by the original author to cause that reader to make an incorrect decision.

operand name context

A study by Jones[686] asked subjects to parenthesize an expression, containing two binary operators, to reflect the relative precedence of the operators. The name of the middle operand was chosen to suggest that it bound to one of the operators (either the correct or incorrect one). For instance, in `a+flags&c` a reader may be influenced to believe (incorrectly) that `flags` is ANDed with `c` and the result added to `a`. The results showed that when the name of the operand matched the context of the correct operator, 76.3% of answers were correct; when the name of the operand matched the context of the incorrect operator, 43.4% of answers were correct.

5.4.1 Language

human language characteristics 792

Natural language issues such as word order, the use of suffixes and prefixes, and ways of expressing relationships, are discussed elsewhere.

In written prose, use of a word that has more than one possible meaning, *polysemy*, can usually be disambiguated by information provided by its surrounding context. The contexts in which an identifier, containing a polysemous word, occur may not provide enough information to disambiguate the intended meaning. The discussion on English prepositions provides some examples.

Prepositions 792

5.4.1.1 Word neighborhood

neighborhood identifier 792

Word neighborhood effects have been found in a number of contexts. They occur because of similarities between words that are familiar to a reader. The amount of familiarity needed with a word before it causes a neighborhood effect is not known. Studies have found that some specialist words used by people working in various fields show frequency effects. The study described next shows a word neighborhood effect— the incorrect identification of drug names.[792.13]

words domain knowledge 792

A study by Lambert, Chang, and Gupta[799] investigated drug name confusion errors. Drug names and their U.S. prescription rates were used, the assumption being that prescription rate is a good measure of the number of times subjects (forty-five licensed, practicing pharmacists) have encountered the drug name. Subjects saw a drug name for three seconds and were then asked to identify the name. The image containing each name had been degraded to a level comparable to that of a typewritten name received through a fax machine with a dirty print cartridge that was running out of ink.

The results found that each subject incorrectly identified 97 of 160 drug names (the degraded image of the drug name being responsible for the very high error rate of 60.6%). Both the frequency of occurrence of drug names and their neighborhood density were found to be significant factors in the error rate (see Figure 792.23). Neighborhood frequency was not a significant factor.

neighborhood identifier 792

An analysis of the kinds of errors made found that 234 were omission errors and 4,128 were substitution errors. In the case of the substitution errors, 63.5% were names of other drugs (e.g., *Indocin*® instead of *Indomed*®), with the remaining substitution errors being spelling-related or other non-drug responses (e.g., *Catapress* instead of *Catapres*®). Figure 792.24 shows the number of substitution errors having a given edit distance from the correct response.

All character sequences ever encountered by a reader can potentially have a word-frequency effect. The character sequences most familiar to developers are those of their native language.

6 Usability

identifier usability

Identifier usability is in the eye (and life experienced mind) of the beholder. The original author of the source, subsequent maintainers of that source, and the managers responsible for products built from the source are likely to have different reasons for reading the source and resources (time and past experience) available to them. Cognitive effort minimization and speed/accuracy trade-offs are assumed to play an important role in identifier usability. As well as discussing visual, acoustic, and semantic factors, this subsection also covers

identifier developer interaction 792

cognitive effort 0

effort vs. accuracy decision making 0

[792.13] Errors involving medication kill one person every day in the U.S., and injure more than one million every year; confusion between drug names that look and sound alike account for 15% to 25% of reported medication errors.

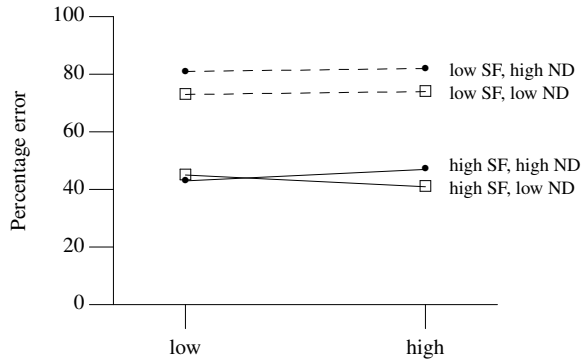


Figure 792.23: Error rate at low and high neighborhood frequency. Stimulus (drug name) frequency (SF), neighborhood density (ND). Adapted from Lambert, Chang, and Gupta.[799]

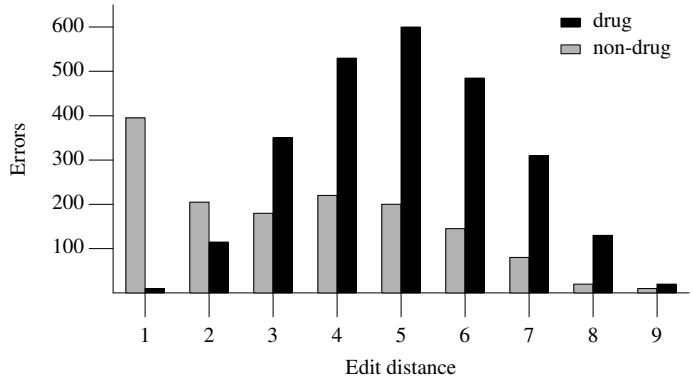


Figure 792.24: Number of substitution errors having a given edit distance from the correct response. Grey bars denote non-drug-name responses, while black bars denote responses that are known drug names. Based on Lambert, Chang, and Gupta.[799]

the use of cognitive resources, initial implementation versus maintenance costs (an issue common to all coding guideline), and typing.

0 coding guidelines importance

Management may show an interest in the spelling used for identifiers for a number of reasons. These reasons, which are not discussed further here, include:

- Some vendors provide interfaces via callable functions or accessible objects, making the names of identifiers visible to potentially millions of developers. Or, a company may maintain internal libraries that are used across many development projects (a likely visibility in the hundreds or perhaps low thousands of developers, rather than millions). In this case customer relation issues are often the most significant factor in how identifier spellings are chosen.
- Project control (or at least the appearance of), often involves the creation of identifier naming guideline documents, which are discussed elsewhere. Other forms of project control are code reviews. These reviews can affect identifier naming in code whether it is reviewed or not, people often use a different decision strategy when they know others will be evaluating their choice.

792 identifier other guideline documents
0 code reviews
0 justifying decisions

6.1 C language considerations

For the majority of identifiers, scope is the most important attribute in deciding their usage patterns:

400 scope

- *local scope.* This is the most common form of identifier usage, both in terms of number of identifiers defined (see Table 439.1) and number of occurrences in the visible source of function definitions.

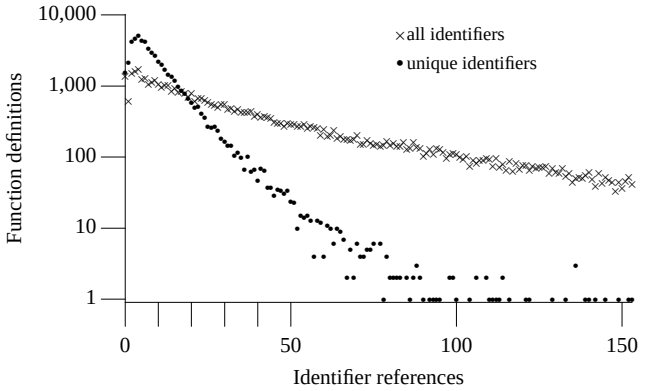


Figure 792.25: Number of identifiers referenced within individual function definitions. Based on the translated form of this book's benchmark programs.

Individual identifier usage is restricted to a relatively small subset of the source code. It is possible for local identifiers in different parts of the source code to share the same name, yet refer to different entities. Identifiers in local scope can be used and then forgotten about. Individually these identifiers may only be seen by a small number of developers, compared to those at global scope.

- *global scope.* While use of this kind of identifier may be restricted to a single source file, it often extends to the entire source code of a program. While, on average, they may be individually referenced more often than individual local identifiers, these references tend to be more widely scattered throughout the source code. Although it may be possible to use and then forget about some identifiers at global scope, it is much more likely that they will need to be recalled, or recognized, across a wider range of program source. They are also much more likely to be used by all the developers working on a project.
- *header file contents.* Over time developers are likely to learn the names of identifiers that appear in included headers. Only identifiers appearing in the most frequently read functions will be familiar. In the early stages of learning about a function, developers are likely to think of identifiers in terms of their spelling; they have yet to make automate the jump to their semantic meaning. How does this impact a reader's analysis of expressions and sequences of statements? It is likely that while being analyzed their component parts will be held in short-term memory.

expressions 940

The majority of references to identifiers occur within expressions and most declarations declare identifiers to be objects (see Table 439.1).

6.2 Use of cognitive resources

identifier cognitive resource usage
attention 0
cogni- 0
tive effort
developer 0
differences

The cognitive resources available to developers are limited. An important aspect of usability is making the optimum use of the available cognitive resources of a reader. Overloading the available resources can lead to a significant increase in the number of errors made.

Different people have different amounts of cognitive resources at their disposal. A general discussion of this issue is given elsewhere. This subsection discusses people's cognitive performance characteristics from the perspective of processing character sequences. While recognizing that differences exist, it does not discuss them further.

A study by Hunt, Lunneborg, and Lewis[608] investigated various performance differences between subjects classified as either *low verbal* or *high verbal*. The Washington Pre-College Test was used to measure verbal ability (a test similar to the Scholastic Achievement Test, SAT). In a number of experiments there was a significant performance difference between low and high verbal subjects. In some cases, differences in performance were only significant when the tests involved existing well-learned patterns. For instance, high

verbals performed significantly better than low verbals in remembering written sequences of syllables when the sequences followed English usage patterns; the performance difference was not very large when nonsense syllable sequences were used.

6.2.1 Resource minimization

identifier resource minimization
0 automatization

Minimizing the amount of resources needed to accomplish some goal is an implicit, if not explicit, human aim in most situations. Studies have found that individuals and groups of people often minimize their use of resources implicitly, without conscious effort.

792 identifier developer interaction
0 developer differences

In the context of this coding guideline resource minimization is a complex issue. The ways in which developers interact with identifiers can vary, as can the abilities (resources available) to individual developers, and management requirements target particular developers (e.g., having certain levels of experience with the source, or having particular cultural backgrounds).

Zipf's law

Zipf noticed a relationship between the frequency of occurrence of some construct, created by some operation performed by people, and the effort needed to perform them. He proposed an explanation based on the principle of least effort. What has become known as Zipf's law[1514] states a relationship between the rank and frequency of occurrence of some construct or behavior. Perhaps its most famous instantiation relates to words, $r = C/f_r$— where r is 1 for the most frequently occurring word, 2 for the second most frequently occurring, and so on; f_r is the number of times the word of rank r occurs; and C is a constant. According to this law, the second most common word occurs half as many times as the most commonly occurring word (in English this is *the*), the third most common occurs $2/3$ times as often as the second most common, and so on.

Zipf's law has been found to provide a good approximation to many situations involving a cost/effort trade-off among different items that occur with varying degrees of frequency. Further empirical studies[1115] of word usage and theoretical analyses have refined and extended Zipf's original formulation.

However, while it is possible to deduce an inverse power law relationship between frequency and rank (Zipf's law) from the principle of least effort, it cannot be assumed that any distribution following this law is driven by this principle. An analysis by Li[853] showed that words in randomly generated texts (each letter, including the space character, being randomly selected) exhibit a Zipf's law like frequency distribution.

The relatively limited selection pressure on identifier spelling (the continued existence of source code does not depend on the spelling of the identifiers it contains and developers are rarely judged by the identifier spelling they create) does not necessarily mean that identifier spellings don't evolve. A study by Kirby[738] found that languages can evolve simply through the dynamics of learning, no selection of learners (i.e., killing off those that fail to reach some minimum fitness criteria) is needed.

A plot of the rank against frequency of identifier spellings (see Figure 792.26) shows a good approximation to a straight line (the result expected from measuring behavior following Zipf's law). Your author cannot provide a reasonable explanation for this behavior.[792.14]

6.2.2 Rate of information extraction

identifier information extraction
0 developer computational power

Reading a character sequence requires identifying each of the characters. The results of a study by Miller et al. found that the amount of information, about a nonword, recalled by subjects was approximately proportional to the time interval it was visible to them. However, the unit of perception used by subjects was not the character. Subjects made use of their knowledge of native language character ordering relationships to chunk sequences of characters into larger perceptual units. A number of studies have suggested various candidates (e.g., syllables, graphemes) for the perceptual reading unit, and these are also described here.

A study by Miller, Bruner, and Postman[936] measured the amount of information subjects remembered about a briefly presented nonword. The nonwords were constructed so as to have different orders of approximation to existing English words (see Table 792.18). Subjects saw a single nonword for a duration of 10, 20, 40, 100, 200, or 500 ms. They then had to write down the letters seen and their position in the nonword (using an answer sheet containing eight blank squares).

[792.14] An explanation for why a ranking of cities by their population follows Zipf's law has been provided by Gabaix,[465] who showed it to be a statistical consequence of individual city population growth following Gibrat's law.

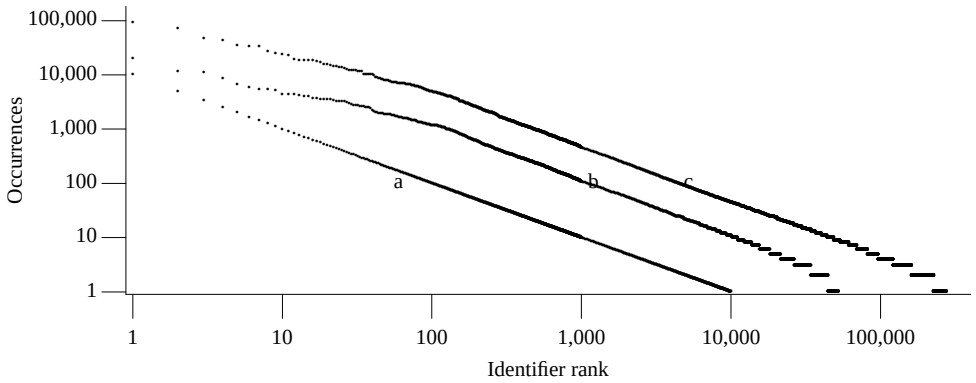


Figure 792.26: Identifier rank (based on frequency of occurrence of identifiers having a particular spelling) plotted against the number of occurrences of the identifier in the visible source of (b) Mozilla, and (c) Linux 2.4 kernel; (a) is a distribution following Zipf's law with the most common item occurring 10,000 times. Every identifier is represented by a dot. Also see Figure 1896.4.

Table 792.18: Examples of nonwords. The 0-order words were created by randomly selecting a sequence of equally probable letters, the 1-order words by weighting the random selection according to the probability of letters found in English words, the 2-order words by weighting the random selection according to the probability of a particular letter following the previous letter in the nonword (for English words), and so on. Adapted from Miller[936]).

0-order	1-order	2-order	4-order
YRULPZOC	STANUGOP	WALLYLOF	RICANING
OZHGPMTJ	VTYEHULO	RGERARES	VERNALIT
DLEGQMNW	EINOAASE	CHEVADNE	MOSSIANT
GFUJXZAQ	IYDEWAKN	NERMBLIM	POKERSON
WXPAUJVB	RPITCQET	ONESTEVA	ONETICUL
VQWVBIFX	OMNTOHCH	ACOSUNST	ATEDITOL
CVGJCDHM	DNEHHSNO	SERRRTHE	APHYSTER
MFRSIWZE	RSEMPOIN	ROCEDERT	TERVALLE

The results (see Figure 792.27) show a consistent difference among the order of approximation for all presentation times. Miller et al. proposed that subjects had a fixed rate of information intake. The performance difference occurred because the higher-order letter sequences had a lower information content for the English language speaking subjects. Had the subjects not been native English speakers, but Japanese speakers for instance, they would have had no knowledge of English letter frequency and the higher-order letter sequences would have contained just as much information as the lower-order ones.

This study (reproduced by Baddeley[74] using spoken rather than visual presentation of letters) shows that developers will need more time to process identifier spellings having a character sequence frequency distribution that does not follow that of their native language. In those cases where source is quickly scanned, a greater number of characters in a sequence (and their positions) are available for recall if they have frequency distribution of the readers' native language.

If individual characters are not the unit of perception and recall used by readers when processing words, what is? The following are a number of proposals:

- A study by Spoehr and Smith[1272] asked subjects to identify some of the letters in a briefly presented letter sequence (the subjects did not know which letters until after they had seen the letter sequence). The results showed that subjects' perceptual accuracy for a word is correlated with the number of recoding steps needed to convert it into speech. For instance, the letter sequences *LSTB*, *BLST*, *BLOST*, and *BLAST* are a sequence not matching English rules, matching English rules but omitting a vowel, a pronounceable nonword, and a word, respectively. They are reproduced correctly in 66%, 70%, 78%, and 82% of cases, respectively. The results are consistent with the letters of a word first being parsed

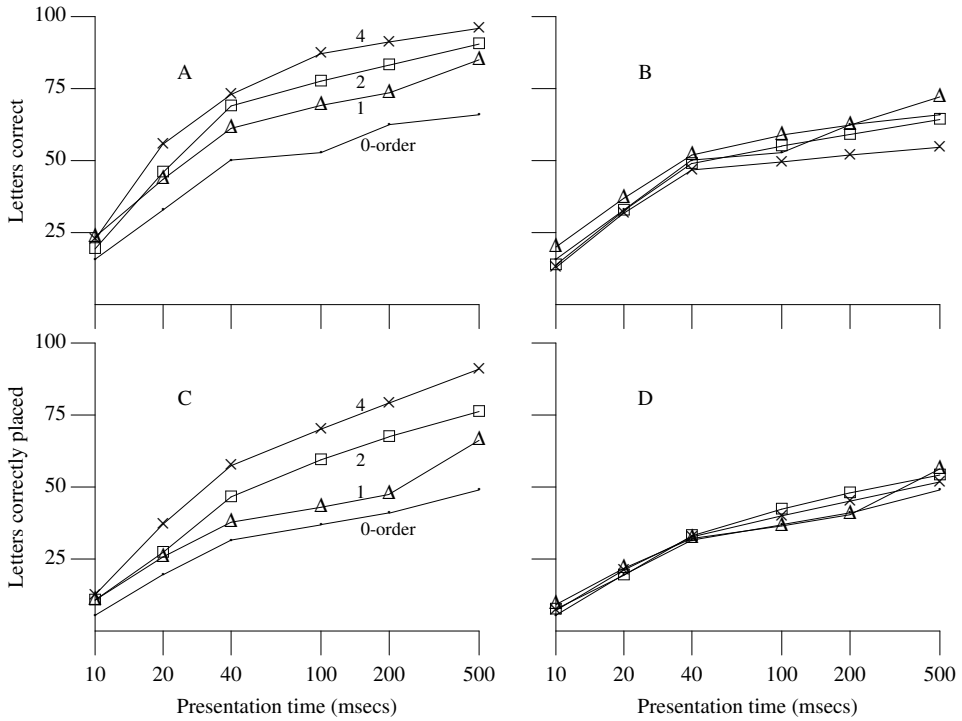


Figure 792.27: Number of correct letters regardless of position (A), and number of correct letters placed in the correct position (C). Normalizing for information content, the corresponding results are (B) and (D), respectively. Plotted lines denote 0-, 1-, 2-, and 4-order approximations to English words (see Table 792.18). Adapted from Miller, Bruner, and Postman.[936]

into syllables.

- A study by Rey, Ziegler, and Jacobs[1160] asked subjects (English and French, using their respective native language) to specify whether a letter was contained within a briefly displayed word. The results found that the response time was longer when the searched for letter was contained within a multi-letter grapheme (phoneme similarity was taken into account). These results are consistent with graphemes being the perceptual reading unit.

792 grapheme

6.2.3 Wordlikeness

The extent to which a nonword is similar to words belonging to a particular natural language is called its *wordlikeness*. Wordlikeness is a cognitive resource in that it represents a person's accumulated reading experience. Wordlikeness has a strong influence on how quickly and accurately nonwords are processed in a variety of tasks.

identifier wordlikeness

How is *wordlikeness* measured? Coleman and Pierrehumbert[253] trained a general phonological word grammar on a list of English words whose phonology was known. The result was a grammar whose transition probabilities correspond to those of English words. This grammar was used to generate a list of letter sequences and subjects were asked to judge their *wordlikeness*. The results showed that the log probability of the phonological rules used in the generation of the complete letter sequence had a high correlation with subjects' judgment of wordlikeness. A study by Frisch, Large, and Pisoni[455] replicated and extended these results.

792 phonology

A number of studies have found differences in people's performance in tasks involving words or nonwords, including:

word nonword effects

- *Visual comparison of words and nonwords.* Studies of performance in comparing two-letter sequences

792 character sequence similarity

have found that response time is faster when the letter sequence represents a word rather than a nonword.

- *Naming latency.* A study by Weekes[1451] found that naming latency for high-frequency words was not significantly affected by the number of letters (between three and six), had some affect for low-frequency words, and had a significant affect for nonwords (this study differed from earlier ones in ensuring that number of phonemes, neighborhood size, bigram frequencies, and other linguistic factors were kept the same). The error rate was not found to vary with number of letters.
- *Short-term memory span.* A study by Hulme, Maughan, and Brown[606] found that subjects could hold more words in short-term memory than nonwords. Fitting a straight line to the results, they obtained:

$$word\ span = 2.4 + 2.05 * speech\ rate \quad (792.1)$$

$$nonword\ span = 0.7 + 2.27 * speech\ rate \quad (792.2)$$

They concluded that information held in long-term memory made a contribution to the short-term memory span of that information.

Wordlikeness may involve more than using character sequences that have an n-th order approximation to the target language. For instance, readers of English have been found to be sensitive to the positional frequency of letters within a word.[902] Using an averaged count of letter digram and trigram frequencies[1265] to create nonwords does not always yield accurate approximations. Positional information of the letters within the word[1266] needs to be taken into account.

Experience shows that over time developers learn to recognize the *style* of identifier spellings used by individuals, or development groups. Like the ability to recognize the *wordlikeness* of character sequences, this is another example of implicit learning. While studies have found that training in a given application domain affects people's performance with words associated with that domain, how learning to recognize an identifier naming *style* affects reader performance is not known.

letter patterns 0
implicit learning
words 792
domain knowledge

6.2.4 Memory capacity limits

identifier
STM required

This subsection discusses short-term memory limitations. Long-term memory limitations are an indirect cause of some of the usability issues discussed under other subsection headings in this usability section. The issue of some identifiers being available in LTM, or migrating to LTM during the process of comprehending source code, is not considered further here.

memory 0
developer
phonological loop

Short-term memory has a number of components, each having different capacity characteristics. The one of immediate interest here is the phonological loop, that is capable of holding approximately two seconds of sound.

Identifiers appear in a number of different source code constructs, most commonly in expressions (the contents of comments are not considered here). When reading an expression, the identifiers it contains are components of a larger whole. One method of reducing the possibility of readers exceeding their short-term memory limit when attempting to comprehend expressions is to minimize the time taken to say (the internal spoken form in a persons head) each identifier. However, short-term memory capacity is not always the main consideration in selecting an identifier spelling.

optimal 792
spelling
identifier

Measuring the amount of time occupied by the verbal form of a word in short-term memory is a nontrivial task[977] (e.g., is it the time taken to say the word in isolation, say the same word repeatedly, or say the word in the context of a list of other words; where *say* is the form spoken internally in the mind, not the spoken form that vibrates the air and can be heard by others).

Calculating the short-term memory requirements needed to hold information on words represented using a logographic writing system is more complicated than for alphabetic writing systems. A study by Hue and Erickson[603] found that literate Chinese subjects represented frequently occurring Chinese characters in verbal form, while low frequency characters were held in visual form.

logographic 792

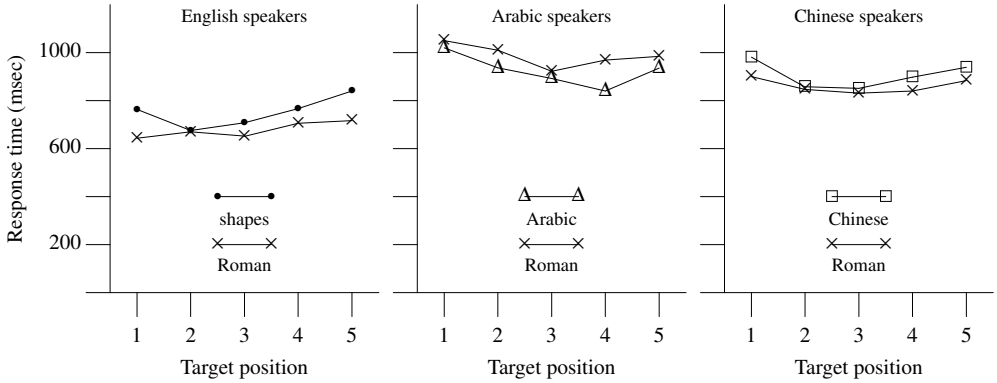


Figure 792.28: Mean response time (in milliseconds) for correct target detection as a function of the position of the match within the character sequence. Adapted from Green and Meara.[518]

6.3 Visual usability

The three main, identifier-related visual operations performed by developers— detailed reading, skimming, and searching — are discussed in the following subsections. The first subsection discusses some of the visual factors involved in extracting information from individual words. The general subject of human visual processing is discussed elsewhere.

770 vision early

6.3.1 Looking at a character sequence

In what order do people process an individual word? Do they start at the leftmost character and move their eyes successively to the right? Do they look at the middle of the word first and progressively work outwards? Do they look at the start and end of the word first, before looking at the rest of the word, or some other order of processing the character sequence?

- A study by Green and Meara[518] investigated the effects of orthography on visual search. Subjects (native English, Spanish, Arabic, or Chinese speakers) were asked to search for a character (Roman letter, Arabic character, or Chinese logograph) in a sequence of five characters (of the same kind). None of the results showed any trade-off between speed and accuracy (which varied between 5–8%). The results (see Figure 792.28) for English and Spanish speakers were very similar— an upward sloping *M* response curve when searching letters and a *U* response curve when searching shapes (with the response slowing as the match position moves to the right). For Arabic speakers there was a *U* response curve for both Arabic and Roman characters (with the response slowing as the match position moved to the left; Arabic is read right-to-left). For Chinese speakers there was a *U* response curve for both Chinese and Roman characters (there was no left or right position dependency).

  792 orthography

  An earlier study[519] comparing the performance of English children and adults found that performance improved with age and that left-to-right processing became more established with age.

- A study by Chitiri and Willows[230] compared how readers of English and Greek paid attention to different parts of a word. Greek primarily uses inflections to denote semantic and syntactic relationships. For instance, one or more characters at the end of a noun can indicate gender, number (singular, plural), and the case (nominative, genitive, accusative, vocative). These letters at the end of a word carry important information, and it is to be expected that experienced readers of Greek will have learned to pay more attention to the ends of words than readers of languages that are not so inflective, such as English. The results showed that Greek readers tended to pay more attention to the ends of words than English readers.

  identifiers Greek readers

- A study by James and Smith[664] asked subjects to search a string of letters for a designated letter. The results showed that for words, in certain cases, vowels were located more quickly than consonants

(there was no difference in performance for nonwords). It was proposed that the difference in performance was caused by the position of vowels in words being more predictable than consonants in English. Subjects were using their knowledge of English spelling to improve their search performance. Searching words in all uppercase or all lowercase did not affect the results.[663]

- A review by Lukatela and Turvey[873] discussed research involving Serbo-Croatian readers. This language has two alphabets, one based on a Cyrillic alphabet and the other on a Roman one, and before the breakup of Yugoslavia schoolchildren in the eastern half of the country learned the Cyrillic alphabet first, followed by the Roman; this order was reversed for schoolchildren in the western half of the country. Some letters were common to both alphabets but had different pronunciations in each. For instance, *potop* could be pronounced */potop/*, */rotop/*, */potor/*, or */rotor/* (two of which represented words— *deluge* in the Roman form, or *rotor* in the Cyrillic form).

- A study by Herdman, Chernecki, and Norris[566] measured subjects' response time and error rate when naming words presented in either lowercase or cAsE aLtErNaTeD form. The words also varied in being either high/low frequency or having regular/irregular spellings. Case alternation slowed response time by approximately 10%. However, it almost doubled the error rate for regularly spelled words (1.8% vs. 3.5% for high-frequency and 5.3% vs. 8.5% for low-frequency) compared to use of lowercase. The cAsE aLtErNaTiOn used in this study is probably more extreme than that usually seen in identifier spellings. As such, it might be considered an upper bound on the performance degradation to be expected when case alternation is used in identifier spelling. A study by Pring[1124] investigated subjects' performance when presented with words using different case. Subjects saw letter sequences such as *CHurCH* and *ChuRCH*. The results showed an increase in error rate (4.4% vs. 2.7%) when a difference in case occurred across grapheme boundaries (a British English speaker might divide *CHURCH* into the graphemes *CH*, *UR*, and *CH*). No difference was found in the error rate for nonwords.

grapheme 792

6.3.2 Detailed reading

Several studies[733] have found that people read prose written using lowercase letters more quickly (approximately 7%) than prose written using uppercase letters. There are a number of reasons for this, including: (1) when proportional spacing is used, a greater number of lowercase characters, compared to uppercase, fall within the visual field allowing larger chunks to be processed per saccade;[108] (2) words appearing in lowercase have a more distinctive shape to them, which is information that enables readers to make more accurate guesses about the identity of a word; and (3) readers have had more practice reading lowercase.

Studies of subjects' eye movements while reading have found that irregular or salient letter sequences at the beginning of a word[610, 1431] cause the eye's initial landing position to be closer to the beginning of the word. However, the initial landing position was not affected by having the salient letters (those needed to disambiguate a word from other words) in the second half of a long word (Finnish words containing 10 to 13 characters).[611] Both of these findings are consistent with the Mr. Chips model of eye movement. Word predictability has been found to have little influence on the initial landing position.[1148]

Mr. Chips 770

There are many differences between reading prose and reading source code. For instance, prose often has a narrative style that allows readers to progress through the material sequentially, while source code rarely has a narrative type and readers frequently have to refer back to previously read material. Whether these constant interruptions reduce the performance advantage of lowercase letters is not known. The extent to which any of these letter–case performance factors affect source code reading performance is not known.

6.3.3 Visual skimming

visual skimming

The number of characters in each identifier that appear in the visible source affects the visual appearance of any construct that contains it. As Figure 792.29 shows, the use of relatively long identifiers can affect the visual layout of constructs that reference them. The layout issues associated with this kind of usage are discussed elsewhere.

expression 940 visual layout
declaration 1348 visual layout
statement 1707 visual layout

```
#include <string.h>

#define MAXIMUM_CUSTOMER_NUMBER_LENGTH 13
#define VALID_CUSTOMER_NUMBER           0
#define INVALID_CUSTOMER_NUMBER         1

int check_customer_number_is_valid(char possibly_valid_customer_number[],
                                   int *customer_number_status)
{
int customer_number_index,
    customer_number_length;

*customer_number_status=VALID_CUSTOMER_NUMBER;
customer_number_length=strlen(possibly_valid_customer_number);
if (customer_number_length > MAXIMUM_CUSTOMER_NUMBER_LENGTH)
   {
   *customer_number_status=INVALID_CUSTOMER_NUMBER;
   }
else
   {
   for (customer_number_index=0; customer_number_index < customer_number_length; customer_number_index++)
      {
      if ((possibly_valid_customer_number[customer_number_index] < '0') ||
          (possibly_valid_customer_number[customer_number_index] > '9'))
         {
         *customer_number_status=INVALID_CUSTOMER_NUMBER;
         }
      }
   }
}
```

Figure 792.29: Example of identifier spellings containing lots of characters. Based on an example from Laitinen.[795]

6.3.4 Visual search

identifier visual search

The commonly chosen, by readers, method of locating occurrences of an identifier in the code visible on a display is visual search. Given such behavior, visual search usability might be defined in terms of selecting an identifier spelling that minimize the probability of a search failing to locate an instance of the searched-for target identifier, or incorrectly classifying an instance as the target.

While there have been a considerable number of studies[1046] investigating visual search, most have involved searching for objects of different shapes and colors rather than words.

Visually source code appears as an ordered sequence of lines. In many cases a complete declaration or statement appears on a single line. The reason for searching the source and the kind of information required can affect what is considered to be the best search strategy; for instance, skimming the characters in a similar order to the one used during detailed reading, scanning down the left edge of the source looking for an assigned-to object, or looking at conditional expressions to see where an object is tested. Your author does not know of any studies that have investigated this issue. The following discussion therefore has to been based primarily on studies using character sequences from other task domains:

- A study by Vartabedian[1412] investigated search times for words presented on a CRT display. Subjects were asked to locate a word among a list of 27 words (either all lowercase or all uppercase) arranged as three columns of nine rows (not forming any meaningful phrases or sentences). The average search time was less (approximately 13%) for the uppercase words.
- A study by Phillips[1084] investigated the effect of letter case in searching for words on paper maps. The results showed that performance was best when the words consisted of an uppercase letter followed by lowercase letters (all uppercase resulted in the worst performance). A second study[1085] investigated the eye fixations used in searching for words in maplike visual displays. The order in which subjects fixated words and the duration of the fixation was measured. The visual similarity between the word

being searched for and the other words was varied (e.g., common initial letter, same word shape based on ascenders and descenders, and word length). The results showed that differences in these visual attributes did not affect eye movements— subjects tended to fixate a word, then a word close to it, and so on. (Some subjects worked from the top-down in a zigzag fashion, others worked clockwise or anti-clockwise around the display.) The duration of the fixation was affected by the similarity of the word being searched for. Objects of similar size and color to words were fixated less often than words, unless the resemblance was very close.

- A study by Flowers and Lohr[431] asked subjects to search for words consisting of either familiar English three-letter words or nonword trigrams with similar features in a display. The time taken, and error rate, for subjects to specify whether a word was or was not present in a display was measured. Distractor words with high similarity to the searched-for word were created by permuting the letters in that word (e.g., *BOY*, *BYO*, *OBY*, *OYB*, and *YBO*). Medium-similarity distractor words contained one letter that was the same as the searched-for word and low-similarity shared no letters. The results showed a significant difference in speed of search with high-similarity nonword distractors. In this case word searches were 30% to 50% faster than nonword searches. This performance difference dropped to 10% to 20% when the distractors were medium-similarity nonwords. There was little difference in performance when the distractors had low similarity or were words. The error rates were not found to be dependent on the distractors.

- A study by Karlin and Bower[713] investigated whether subjects could categorize a word semantically before they precisely identified the word. The time taken and error rate for subjects to specify whether a word was or was not present in a display was measured. The searched-for word either belonged to the same category as the other words displayed, or to a different category. For instance, the name of a color, say *PINK*, might be displayed with the names of other colors or the names of trees. The results showed that as the number of distractor words increased subjects increased their use of category information to improve search time performance (searching for a word in a different category was nearly a third faster when searching among six items than when it was in the same category). The error rates were not found to be category-dependent. Karlin and Bower proposed that comparison involved two kinds of tests, performed in parallel— a categorization test and a perceptual feature test. In those cases where a comparison requires a lot of perceptual resources because two items are perceptually similar, the categorization test may complete first. Flowers and Lohr explained their results in terms of categories, words, and nonwords. Searching for a word amongst nonword distractors that share the same letters imposes a high perceptual load because of overlap of features, and the categorization comparison completes first. When the distractors share a single letter, the difference is less pronounced.

When searching for an identifier, it is possible that mistakes will be made (either failing to locate an identifier that is present, or incorrectly matching the wrong identifier). The issue of identifier confusability is discussed elsewhere.

identifier confusability 792

6.4 Acoustic usability

A number of factors contribute toward the acoustic usability of a character sequence. Memory and confusability acoustic factors are discussed in earlier subsections. Here, generating a pronunciation for a character sequence and meanings suggested by sounds (phonetic symbolism) are discussed.

phoneme 792

Phonological codes (the basic sounds of a language) have been found to play an important role in accessing the semantic information associated with words written in nonlogographic scripts (some researchers believe they are obligatory,[433, 1403] while others believe there is a single mechanism, e.g., connectionist models[1094]). It was once thought that, because of their pictorial nature, logographs did not evoke phonological codes in readers. Studies[1330] have found that phonographic codes also play a role in reading logographic scripts.

logographic 792

6.4.1 Pronounceability

word pronounceability

All character sequences can be pronounced in that the individual characters can be pronounced one at a time.

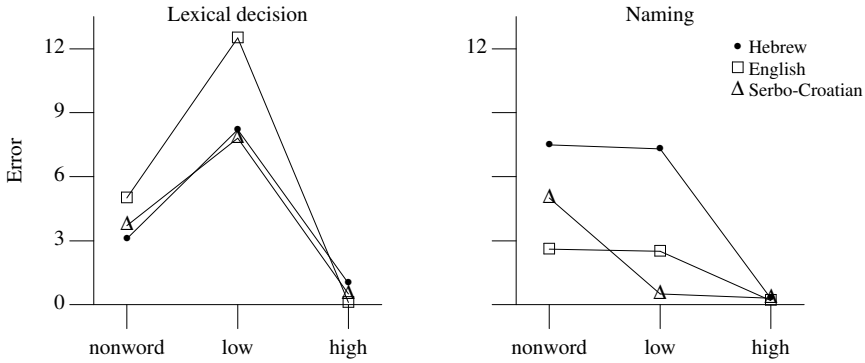


Figure 792.30: Error (as a percentage of responses) for naming and lexical decision tasks in Hebrew, English, and Serbo-Croatian using high/low frequency words and nonwords. Adapted from Frost, Katz, and Bentin.[457]

The term *pronounceable* is commonly applied to character sequences that are wordlike and can be converted to a spoken form using the grapheme-to-phoneme conventions of a natural language familiar to the reader. In some languages (e.g., Japanese kana and Greek) virtually any combination of characters is pronounceable. This is because each character represents an individual sound that is not significantly affected by adjacent the characters.

792 identifier wordlikeness

'Twas brillig, and the slithy toves Did gyre and gimble in the wabe: All mimsy were the borogoves, And the mome raths outgrabe. Lewis Carroll

In this subsection pronounceability is measured in terms of the ease with which a reader is able to convert a sequence of characters to a spoken form (pronounceability could also be defined in terms of minimizing information content). Whether the spoken form used by the reader is the one intended by the author of the source code is not of concern here.

Abbreviating words removes what appear to be redundant characters. However, these characters are needed by readers if they are to recognize the graphemes their prior experience has trained them to expect (unless the reader recognizes the abbreviation and internally uses the word it represents). Abbreviations thus reduce pronounceability.

792 characters mapping to sound
792 identifier information extraction
792 abbreviating identifier

Experience shows that developers use a number of techniques to generate a spoken representation of character sequences. In some cases the sounds of the individual characters are used. In other cases developers mentally add characters (vowels and sometimes consonants) to what would otherwise be a nonpronounceable nonword (for abbreviations this is often the unabbreviated word); the sound used evolves over time, particularly when the character sequence is used in a spoken form between developers. Some developers simply take pleasure in inventing sounds for character sequences.

- A study by Frost, Katz, and Bentin[457] investigated the influence orthographic depth had on visual word recognition. Subjects were native speakers of Hebrew (a very deep orthography), Serbo-Croatian (a very shallow orthography), and English (an orthography somewhere between the two). The tasks involved word naming or making a word/nonword lexical decision and the response times and error rates were measured. The results were consistent with the idea that readers of shallow orthographies generate the pronunciation directly from the words, while for readers of deeper orthographies it is derived from an internal lexicon. The error rates are shown in Figure 792.30

  792 orthography

  Brain imaging studies[1063] have found that Italian speakers (shallow orthography) activate different parts of their brain while reading compared to English speakers (deep orthography).

- A study by Lange and Content[808] analyzed the grapheme-to-phoneme correspondence of French words. (The results confirmed the view that this was quite predictable; phoneme to grapheme correspondence was not analyzed and is considered to be much less predictable.) They then used the results of this analysis to measure French speakers' performance when reading words aloud. The selected words had either low/high grapheme frequency (number of occurrences in the corpse, independent of phoneme mapping) or low/high grapheme entropy (a measure of the number of different phonemes,

and their frequency, a particular grapheme could be mapped to). They found that time to name a word and error rate did not vary significantly with grapheme frequency. However, there was a significant difference in error rate (but not time to name the word) when comparing words with a low/high grapheme entropy. This provided experimental evidence that more naming errors are made for words containing graphemes having many possible pronunciations than those having fewer possible pronunciations.

- A study by Rey, Jacobs, Schmidt-Weigand, and Ziegler[1159] asked subjects to identify words (using matching native English and French words and subjects) containing five letters. These five-letter words contained either three, four, or five phonemes (e.g., for English: TEETH */tiT/*, BLEAT */blit/*, or BLAST */bl#st/* respectively). Subjects' response time and error rate were measured. The results showed that subjects' performance improved (slightly faster response time and decreased error rate) as the number of phonemes increased (except for high-frequency French words). A study by Rastle and Coltheart[1145] found the same behavior for nonwords in both their DRC model of word naming and human subjects. The results from the DRC model suggested that reader expectations were causing the difference in performance. As each letter was successively processed, readers used it and previous letters to create a phoneme. In many cases these phonemes contained a small number of letters (perhaps just one) and readers started processing based on an expected common case. For instance, the letter *P* often represents the phoneme */p*; however, the following letter may create a multi-letter phoneme, and the previous processing on creating a pronunciation needs to be partially undone and processing started again (e.g., when the letter *P* is followed by *H*, the phoneme */f/* needs to be used). In the preceding studies it is not that the five phoneme letter sequences are processed more quickly, but that the three phoneme sequences are significantly slowed by additional cycles of processing, undoing, and reprocessing.

Word recognition 792
models of

- A study by Stanovich and Bauer[1284] showed that the regularity of spelling-to-sound correspondence affected performance; for instance, the regular pronunciation of *_INT* occurs in *MINT*, *HINT*, *DINT*, and an irregular pronunciation in *PINT*. Their study showed that regularity had some impact on the time taken to name a word (552 ms vs. 570 ms) and on a lexical decision task (623 ms vs. 645 ms).
- A study by Stone, Vanhoy, and Orden[1303] showed that not only did spelling-to-sound regularity (feed-forward consistent) affect performance, but sound-to-spelling regularity (feedback consistency) was also a factor. Visual word perception is a two-way street. For instance, the sound */_ip/* can have either of the spellings that occur in *HEAP* and *DEEP*. The results showed, in a lexical decision task, an error rate of 3.9% for feedback-consistent and 9.8% for feedback-inconsistent words. There was no difference in error rate for nonwords.

6.4.1.1 Second language users

The following discussion and studies are based on having English as the second language. The results point to the conclusion that encoding strategies used by a person in their first language are transferred to English. This behavior can result in a different phonological encoding being formed, compared to the set of encodings likely to be used by native English speakers, when an unknown character sequence is encountered. The issue of pronouncing characters unfamiliar to the reader is discussed elsewhere.

phonology 792
reading 792
characters
unknown to reader

- A study by Holm and Dodd[587] investigated how phonological awareness skills acquired while learning to read and write a first language were transferred to learning to read and write a second language. The subjects were students at an Australian university who had first learned to read and write their native language in China, Hong Kong, Vietnam, and Australia. The characters learned by the Chinese and Hong Kong subjects were the same— logographs representing a one-syllable morpheme. The Vietnamese and Australian subjects had been taught using an alphabetic writing system, where a mapping to phonemes existed. As an aid to the teaching of reading and writing, China introduced an alphabetic system using Latin symbols called *pinyin* in 1959. No such system had been used by the subjects from Hong Kong. One of the tasks involved creating spoonerisms from pairs of written

Spoonerisms

words (e.g., *dark ship* ⇒ *shark dip*). This task requires segmenting words based on phonological rules not letter rules. The results of the study showed that the performance of Hong Kong subjects was significantly worse than the other subjects in these kinds of phonological-based tasks. While they were proficient readers and writers of English, they had no ability to associate a written word with its pronunciation sound unless they had previously been told of the association. Holm and Dodd compared the performance of the Hong Kong students with phonological dyslexics that have been documented in the research literature. Interviews with the Chinese subjects found that one of the strategies they used was to "thinking how to write the word in pinyin". Some of the Hong Kong students reported being able to recognize and use written words by looking up their definitions in a dictionary. However, on hearing one of these words in a lecture, they were not able to make the association to the written form they had previously encountered unless they saw it written down (e.g., in the notes written by the student sitting next to them in class). The results of this study showed that for some developers identifier pronounceability is not an issue because they don't pronounce them.

- A study by Koda[754] investigated the impact of subjects' first language (Arabic, English, Japanese, or Spanish; who had had at least six years of English instruction) on their phonological encoding strategies for various kinds of character sequences. Subjects were shown five character sequences followed by a probe (one of the five character sequences). They had to specify which character sequence followed the probe. The character sequences were either phonologically similar English nonwords, phonologically dissimilar but visibly similar English nonwords, unpronounceable sequence of letters, pronounceable Japanese Kanji (logographs that were either visually similar or dissimilar), unpronounceable Japanese Kanji, or Sanskrit (a logography unfamiliar to all subjects). The results showed that subjects, independent of language background, performed better when phonologically dissimilar English pronounceable nonwords were used (phonological similarity causes interference in working memory) and the performance of Japanese subjects was not significantly affected by the use of unpronounceable English nonwords. The results from the Japanese lists showed that performance among the three non-Japanese subjects did not differ significantly. There was no significant difference between any group of subjects on the Sanskrit lists.

  792 acoustic confusability

- A study by Furugori[464] found that the typical spelling mistakes of Japanese users of English reflected the lack of differentiation, made in Japanese, of some English phonemes (e.g., Japanese has no */θ/* phoneme as in *th*ink, which is heard as */s/*, leading to *thunderstorm* being spelled *sanderstorm*).

6.4.2 Phonetic symbolism

identifier phonetic symbolism

Studies have found that native speakers of English have the ability to guess the meanings of words from unfamiliar languages with greater than chance probability. For instance, a study by Brown, Black, and Horowitz[171] asked subjects (English speakers unfamiliar with the other languages used in the study) to match English antonyms against the equivalent Chinese, Czech, and Hindi pairs of words (58.9%, 53.7%, and 59.6% pairs were correctly matched, respectively). It has been proposed that there exists a universal phonetic symbolism. This proposal implies that sounds tend to have intrinsic symbolic connotations that are shared by humans and that traces of these sound-to-meaning linkages survive in all natural languages.

792 antonym

A study by Koriat[764] found that the stronger a subject's feeling of knowing for a particular antonym pair, the higher the probability of their match being correct (English antonyms were required to be matched against their Thai, Kannada, and Yoruba equivalents).

0 feeling of knowing

The symbolism commonly associated with certain word sounds has been known about for many years; for instance, that words with *K* or *P* sounds are funny (chicken, pickle, cucumber, and porcupine). Advertisers also make use of the sound of a word when creating names for new products. A study by Schloss[1201] showed that 27% of the top 200 brands of 1979 began with *C*, *P*, or *K*; 65% began with *A*, *B*, *C*, *K*, *M*, *P*, *S*, or *T* (with less than a 5% probability of this occurring by chance).

A study by Magnus[886] looked at monosyllabic words in English. The results showed that words containing a given consonant fell within much narrower semantic domains than would be expected if there were no

correlation between phonology and semantics. The term *phonesthemes* was defined to refer to a sound sequence and a meaning with which it is frequently associated. An example of a phonestheme is the English */gl/* in initial position being associated with indirect light.

Table 792.19: Words that make up 19 of the 46 words beginning with the English */gl/* of the monomorphemic vocabulary (Note: The others are: globe, glower, glean, glib, glimmer, glimpse, gloss, glyph, glib, glide, glitter, gloss, glide, glissade, glob, globe, glut, glean, glimmer, glue, gluten, glutton, glance, gland, glove, glad, glee, gloat, glory, glow, gloom, glower, glum, glade, and glen). Adapted from Magnus.[886]

Concept Denoted	Example Words
Reflected or indirect light	glare, gleam, glim, glimmer, glint, glisten, glister, glitter, gloaming, glow
Indirect use of the eyes	glance, glaze(d), glimpse, glint
Reflecting surfaces	glacé, glacier, glair, glare, glass, glaze, gloss

Magnus also asked subjects (predominantly English speakers responding to a Web survey) to provide definitions for made-up words. For many of the words the particular definitions provided where limited to a small number of semantic domains, often with two domains accounting for more than half of the definitions.

6.5 Semantic usability (communicability)

Identifier semantic usability

A commonly specified coding guideline recommendation is that *meaningful* identifiers be used. This subsection discusses the topic of shared semantic associations (the sharing is treated as occurring between the original creator of an identifier spelling and subsequent readers of it), a more technical way of saying *meaningful*.

semantic networks 792

Developers have an existing semantic net in their head, which maps between character sequences and various kinds of information. This net has been continually building up since they first started to learn to read and write (as such, very little of it is likely to be directly related to any computer-related applications or software development). This existing semantic net is of major significance for several reasons, including:

- Making use of existing knowledge enables a small amount of information (an identifier's spelling) to represent a much larger amount of information. However, this is only possible when the original author and the reader share the same associations (or at least the ones they both consider applicable to the circumstances) for the same character sequences.
- Creating new memories is a time-consuming and error-prone process; making use of existing ones can be more efficient.

implicit learning 0

- A large amount of the information that it contains is not explicitly available. Developers often apply it without consciously being aware of the extent to which their decisions are driven by culturally (in its broadest sense) specific learning.

Given the current state of knowledge about the kinds of semantic associations made by groups of people working in a common field, even predicting how developers educated in the same culture might communicate effectively with each other is very difficult, if not impossible. Predicting how developers educated in different cultures might communicate, via identifier spellings, creates even more layers of uncertainty.

However impossible prediction might be at the time of writing, it is a problem that needs to be addressed. The following subsections discuss some of the issues.

6.5.1 Non-spelling related semantic associations

The need to recall information about an identifier is often prompted by its being encountered while reading the source. (A developer may ask another developer for information on an identifier and provide a spoken rendition of its spelling, but this usage is much less common.) The context in which the identifier occurs can often be used to deduce additional information about it. For instance, the identifier is a typedef name (it occurs in a list of declaration specifiers), it has arithmetic type (it occurs as the operand of a binary

multiplication operator), or designates an array or function (it occurs immediately to the left of a particular kind of bracket).

Although often providing important information to readers of the source, these nonspelling-related semantic associations are not discussed further here.

6.5.2 Word semantics

What does it mean to know a word? Richards[1161] gave the following answer, which has been very influential in the field of vocabulary acquisition:

1. The native speaker of a language continues to expand his vocabulary in adulthood, whereas there is comparatively little development of syntax in adult life.
2. Knowing a word means knowing the degree of probability of encountering that word in speech or print. For many words, we also *know* the sort of words most likely to be found associated with the word.
3. Knowing a word implies knowing the limitations imposed on the use of the word according to variations of function and situation.
4. Knowing a word means knowing the syntactic behavior associated with that word.
5. Knowing a word entails knowledge of the underlying form of a word and the derivatives that can be made from it.
6. Knowing a word entails knowledge of the network of associations between that word and the other words in language.
7. Knowing a word means knowing the semantic value of a word.
8. Knowing a word means knowing many of the different meanings associated with the word.

The following discussion involves what was once thought to be a semantic effect on word naming performance— *imagineability*. Subsequent studies have shown that age of acquisition is the primary source of performance difference. This work is discussed here because many sources cite imagineability effects as a word-handling performance issue. A study by Strain, Patterson, and Seidenberg[1304] asked subjects to read aloud words that varied across high/low frequency, regular/irregular spelling–sound correspondence, and high/low imagineability (as judged by 40 members of staff at the author's workplace; examples of high imagineability included *corkscrew* and *sparkling*, while words with low imagineability included *naive* and *presumption*). The results showed that on one case (low-frequency, irregular spelling–sound correspondence low imagineability) the error rates were significantly higher (14–19% vs. 0–3%) than the other cases. It was proposed that the semantic clues provided by imagineability provide a performance benefit that is only significant when processing irregularly spelled low frequency words. However, a later study by Monaghan and Ellis[955] also took a words age of acquisition into account. The results showed that age of acquisition had a significant effect on word naming performance. Once this effect was taken into account the apparent effects of imagineability disappeared. It was pointed out that words learned early in life tend to be less abstract than those learned later. It is age of acquisition that is the primary effect.

792 age of acquisition

6.5.3 Enumerating semantic associations

semantic associations enumerating

How can the semantic associations evoked by a word be enumerated? One method is to enumerate the list of words that are considered to be related, or similar, to it. To this end this subsection discusses some of the algorithms that have been proposed for measuring the semantic relatedness, or similarity, of two words. It is based on the review by Budanitsky.[177] To quote from its opening sentence "Is *first* related to *final*? Is *hair* related to *comb*? Is *doctor* related to *hospital* and, if so, is the connection between them stronger than that between *doctor* and *nurse*?"

The methods of measuring similarity proposed by researchers can be broadly divided into two groups. The context-free methods do not consider the context in which the two words occur. Some compendium of words (e.g., a dictionary or thesaurus) is used to provide the base information. The context-sensitive methods consider the context in which the two words are used.

6.5.3.1 Human judgment

word similarity human judgment

One way to obtain a list of words associated with a particular word is human judgment. Studies of human semantic memory, by cognitive psychologists, often make use of word association norms. However, most of these studies use a small set of words. For instance, a study by Burke, Peters, and Harrold[183] measured associations for 113 words using 80 young (mean age 21.7) and 80 older (mean age 71.6) subjects. There have been two studies, using English speakers, that have collected associations for a large number of words.

A long-term study (it started in 1973) by Nelson, McEvoy, and Schreiber[1006] created the University of South Florida word association, rhyme, and word fragment norm database by collating the nearly three-quarters of a million responses to 5,019 stimulus words from more than 6,000 subjects. Subjects (students at the university) were given lists of words (approximately 100 in a booklet, with 25–30 per page) and asked to write the first word that came to mind that was meaningfully related or strongly associated to each of these words. For example, given *book*, they might write *read*.

Nelson et al. compared their results with those obtained by Kiss, Armstrong, Milroy, and Piper[740] in the UK. They found substantial differences between the results and suggested that these were caused by cultural differences between Florida and the UK. For example, the most frequent responses of the Florida subjects to the word *apple* were *red* and *orange* (the fruit), with *tree* and *pie* being given relatively infrequently.

A study by Steyvers[1298] used the Nelson et al. word association data to build a *Word Association Space* (which was claimed to have psychological relevance). The results showed that this space was a good predictor of similarity rating in recognition memory, percentage correct responses in cued recall, and intrusion rates in free recall.

6.5.3.2 Context free methods

The advantage of using existing sources of word associations, such as dictionaries, is the large amount of information they provide about general word associations. The disadvantages with using these sources of information is that the word associations they contain may not include specific, technical uses of words or even have some of the words being analyzed. The availability of large volumes of written text on the Internet has shown that the primary definitions given to words in dictionaries are often not the definitions commonly used in this material (although publishers are starting to produce dictionaries[257] based on measurements of common word usage). Also a human-written dictionary in computer readable form may not be available (the calculations for the original thesaurus study[965] were done by hand because online material was not available to the authors).

The following are some of the studies that have used context-free methods:

- A study by Kozima and Furugori[769] used a subset of the *Longman Dictionary of Contemporary English*[867] (*LDOCE*) to build a semantic network. This dictionary uses a small (2,851 words) vocabulary to express the meanings of all the other words (more than 56,000) it defines.[792.15] This semantic network was used to calculate the similarity between words using spreading activation. Later work by Kozima and Ito[770] made context-sensitive measurements by performing what they called *adaptive scaling of the semantic space*. They gave the example of *{car, bus}* having the context *vehicle* and being closely associated with *taxi*, *railway*, *airplane*, and so on, while *{car, engine}* had the context *components of a car* and were closely associated with *tire*, *seat*, *headlight*, and so on. A change of context usually resulted in a change of distance between the same word pair.
- Ellman[386] created a tool to detect lexical chains (constructs used in calculating text similarity) in large texts, with the intent of extracting a representation of its meaning. Various kinds of semantic relations between the words classified in *Roget's Thesaurus* (a thesaurus is intended as an aid in finding words that best express an idea or meaning, while a dictionary explains the meaning of words) were used to compute lexical chains between pairs of words appearing in the texts.

[792.15] This feature, along with the publisher making it available to researchers for a small fee, has made LDOCE widely used by language researchers.

6.5.3.3 Semantic networks

semantic networks

A semantic network consists of a set of nodes with connections, and arcs, between them. The nodes representing concepts and the arcs denoting relationships between the nodes (concepts) they connect.

WordNet

WordNet[792.16][416] is a semantic network whose design was inspired by current psycholinguistic theories of human lexical memory. It organizes English nouns, verbs, adjectives, and adverbs into synonym sets (synsets), each expressing one underlying lexical concept (see Table 792.20). Edmonds[377] provides a detailed discussion (and a computational model) of the fine-grained meanings of near synonyms and the differences between them. For instance, all of the WordNet noun synsets are organized into hierarchies. At the top of the hierarchies are the following nine abstract concepts called *unique beginners*:

792 synonym

Table 792.20: WordNet 2.0 database statistics.

Part of Speech	Unique Strings	Synsets	Total Word-sense Pairs
Noun	114,648	79,689	141,690
Verb	11,306	13,508	24,632
Adjective	21,436	18,563	31,015
Adverb	4,669	3,664	5,808
Total	152,059	115,424	203,145

- Entity— that which is perceived or known or inferred to have its own physical existence (living or nonliving)
- Psychological feature— a feature of the mental life of a living organism
- Abstraction— a general concept formed by extracting common features from specific examples
- State— the way something is with respect to its main attributes: *the current state of knowledge*, *his state of health*, *in a weak financial state*
- Event— something that happens at a given place and time
- Act, human action, human activity— something that people do or cause to happen
- Group, grouping— any number of entities (members) considered as a unit
- Possession— anything owned or possessed
- Phenomenon— any state or process known through the senses rather than by intuition or reasoning

The arcs between nouns in synsets are defined by the following relations:

- Antonym— the *complement* relation; words of opposite meaning (e.g., hot-cold)
- Holonymy— the *has a* relation, the opposite of meronymy

antonym

- Hypernymy— the *is a* relation (e.g., plant is a hypernym of tree)
- Hyponymy— the *subsumes* relation, the inverse of hypernymy
- Meronymy— relationship between objects where one is a part of the other (e.g., sleeve is a meronym of coat, dress, or blouse)
- Synonym— word with the same, or nearly the same meaning

synonym

Steyvers and Tenenbaum[1300] investigated the graph theoretic properties of the semantic networks created by WordNet, *Roget's Thesaurus*, and the associative word lists built by Nelson et al. The results showed that they had a small world[25] structure.

[792.16] Probably the most well-known semantic network in linguistics, and it is available for download from the internet.

6.5.3.4 Context sensitive methods

The attributes associated with an unknown word can often be inferred from the context in which it occurs; for instance, the paragraph: "A bottle of tezgüino is on the table. Everybody likes tezgüino. Tezgüino makes you drunk. We make tezgüino out of corn." suggests that *tezgüino* is an alcoholic drink made from corn mash.

Context-sensitive methods obtain their information directly from a corpus of written material. These methods all assume the words occur in text written in a natural language. As such, they are not directly applicable to identifiers in source code. However, they do provide a possible mechanism for automatically obtaining word similarity information for specialist domains (e.g., by processing books and papers dealing with those domains).

The advantages of context-sensitive methods are that they are not limited to the words appearing in some predefined source of information and because the needed information is automatically extracted from a corpus they are sensitive to the associations made. The disadvantage of this method is that the corpus may not contain sufficient occurrences of particular words for an accurate evaluation of their associations to be calculated.

latent semantic analysis

One of the most widely discussed context-sensitive methods is *Latent Semantic Analysis*, LSA.[802] The underlying idea is that the sum of all the contexts in which a given word does and does not appear provides a set of mutual constraints that determines the similarity of meanings of words and sets of words to each other. The process of extracting relations between words starts with a matrix, where each row stands for a unique word and each column stands for a context (which could be a sentence, paragraph, etc.). Each matrix cell holds a count of the number of times the word it represents occurs in the context it represents. Various mathematical operations are performed (to the uninitiated these seem completely disconnected from the problem at hand and for this reason are not described here) on the matrix to yield results (each word is mapped to a vector in an n-dimensional space, where n is usually around 300, and the similarity between two words is calculated from cosine of the angle between their respective vectors) that have been found to effectively model human conceptual knowledge in a growing number of domains.

morphology 792

LSA takes no account of word order ("dog bites man" and "man bites dog" are treated the same way), syntactic relations (no syntactic parsing of the text is performed), or morphology (*fast* and *faster* are treated as different words). This eliminates many of the practical difficulties experienced by other methods. This simplicity, along with the quality of its results has made LSA a popular choice for information-retrieval problems.

An obvious way to try to improve the quality of word similarity measures is to take their syntactic relationships into account. Lin[856] proposed a similarity measure that takes the grammatical relationship between the two words into account. The raw data from which this word information is extracted is a list of sentences. These sentences are broken down into dependency triples, consisting of two words and the grammatical relationship between them, within the sentence. For instance, the triples for the sentence "I have a brown dog" are: (*have* subj *I*), (*I* subj-of *have*), (*dog* obj-of *have*), (*dog* adj-mod *brown*), (*brown* adj-mod-of *dog*), (*dog* det *a*), (*a* det-of *dog*), given two words w and $w^{'}$, and the relation r. The similarity between two words is based on the amount of information contained in the commonality between them, divided by the amount of information in the description of them, and the amount of information, I, contained in the dependency tuple $\parallel w, r, w^{'} \parallel$ is given by:

$$I(w,r,w^{'}) = \log(\frac{\parallel w,r,w^{'} \parallel \times \parallel *,r,* \parallel}{\parallel w,r,* \parallel \times \parallel *,r,w^{'} \parallel}) \tag{792.3}$$

where $*$ matches all words.

Lin obtain his sentences from various online newspapers (a total of 64 million words). An English language parser extracted 56.5 million dependency triples, 8.7 million being unique. There were 5,469 nouns, 2,173 verbs, and 2,632 adjectives/adverbs occurring more than 100 times.

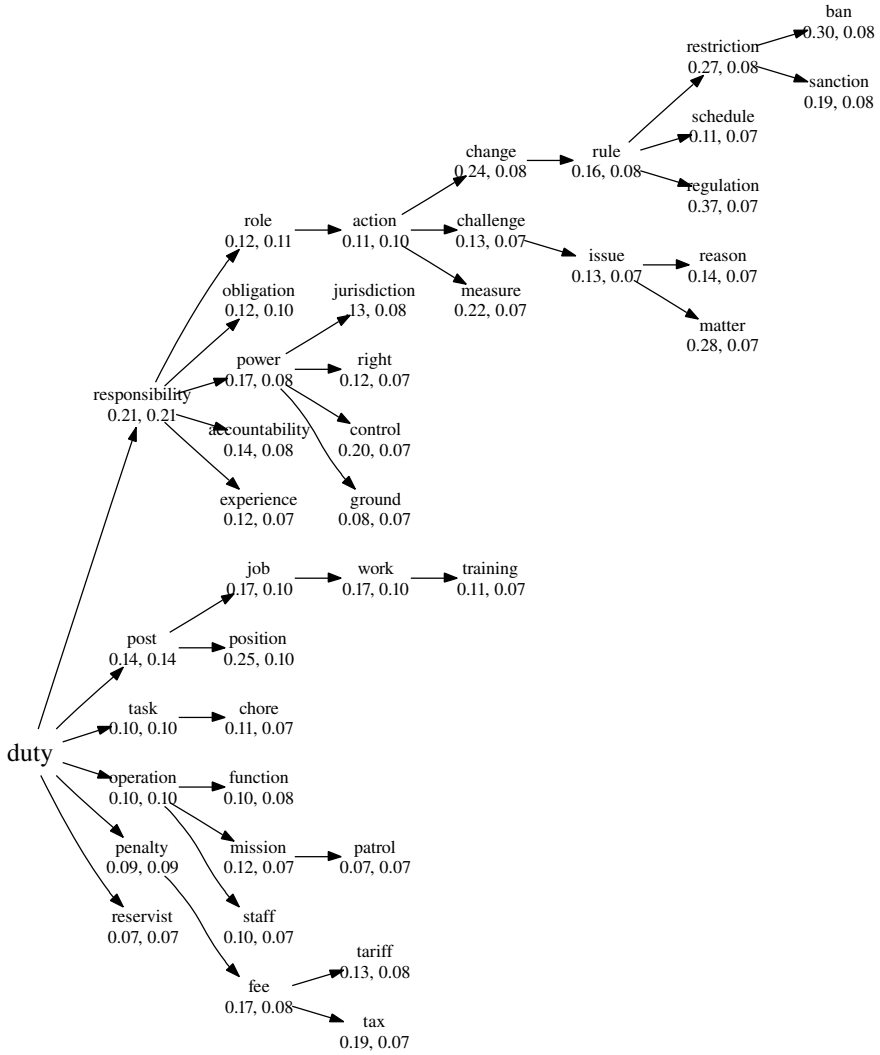


Figure 792.31: Semantic similarity tree for *duty*. The first value is the computed similarity of the word to its parent (in the tree), the second value its similarity to *duty*. Adapted from Lin.[856]

6.5.4 Interperson communication

Identifier spellings provide a delayed, one-way form of human communication. The original author decides on a spelling to use, often with the intent of it denoting meaningful information, and sometime later a second person reads it (code reviews offer an opportunity for other developers to perform the role of future readers, 0 code reviews but they are not usually held for this purpose). This form of communication is significantly different from the collaborative process that underlies most human communication. For instance, a study by Clark and Wilkes-Gibbs[244] showed how two people work together in the creation of agreed-on references (to complex shapes). The results also found that the number of words used decreased over successive trials (rearranging square and triangular paper cards to form complex shapes) as subject pairs learned from each other.

The writers and readers of source code rarely get to take part in a collaborative communications process with each other. Furthermore, the original author may not even have other human readers in mind, when deciding on an identifier spelling. Authors may see themselves as communicating with the computer or communicating with themselves at some future date.

The following two subsections discuss the evolution of terminology groups by use in communicating among themselves and some of the issues involved in two people reaching the same conclusions about the semantic associations of a word or phrase.

6.5.4.1 Evolution of terminology

All but the smallest software development project will have more than one person working on it, although each particular piece of source code often has a single person working on it. While in many cases the number of people who actually write significant amounts of source is usually only a fraction of the total number of people on the project, there is invariably a set of linguistic conventions (including a terminology) that evolves and is shared by a large percentage of the members of a project.

The characteristics of the evolution of linguistic conventions that occur in groups are of interest to the extent that they affect the performance of subsequent readers of the source.

A study by Garrod and Doherty[473] investigated the establishment of linguistic conventions in two different kinds of groups. Two people had to work together to solve a maze game in which the information needed was distributed between them (the only way of communicating the needed information was by questioning and answering each other; responses were recorded by the experimenter). In one group (five pairs) the same two people always worked together, while in the other group everybody (10 people) eventually got to be paired with everybody else in the group. Each subject played nine 10-minute games with the maze randomly generated for each game.

The results showed that isolated pairs of subjects had much higher levels of inter-speaker coordination (measured by categorizing each information exchange that took place between subjects while solving a particular maze, and counting the exchanges in each category) compared to the community pairs during the first few games. However, by the time six games had been played, this situation had reversed, with the community pairs having significantly more inter-speaker coordination.

The way in which the two groups coordinated their descriptions differed. The isolated pairs used descriptions that were specific to a particular position in the solution and specific to themselves. Once the community pairs became established through overlapping interactions, they began to coordinate as a group. The constraints on their descriptions became global ones that apply to communities in general (i.e., they were not able to make use of special cases previously agreed to between two individuals).

If the behavior seen in the Garrod and Doherty study also occurs in development groups, it is to be expected that groups of one or two developers are likely to evolve terminology that is much more tightly bound to their mutually agreed-on way of thinking than larger groups of developers.

6.5.4.2 Making the same semantic associations

The semantic usability of an identifier's spelling might be judged by the degree to which the semantic associations it creates reduces the effort needed to comprehend source code containing it. A randomly selected identifier spelling is extremely unlikely to create semantic associations having this property. Selecting an identifier's spelling to create the necessary associations is the solution. However, the decision on the spelling to use is judged by the semantic associations created in the original author's mind at the time it is selected. It cannot be assumed that the semantic associations of the original author are the ones most relevant to subsequent readers. This subsection discusses some of the issues involved.

Both the authors and readers of an identifier's spelling will make assumptions about how it is to be interpreted.

- *Authors* of source code will make assumptions about the thought process of subsequent readers. Explicit assumptions made might include: degree of familiarity with application domain (e.g., to simplify their task the original authors may have specified to management that competent people need to be hired and then proceed to work on the basis that this requirement will be met), or that they have read the documentation. Implicit assumptions might include: culture (no thought given to alternative cultures), degree of familiarity with the application domain (e.g., is familiar with certain terms such as

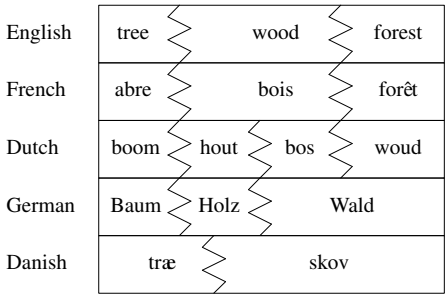


Figure 792.32: The relationship between words for tracts of trees in various languages. The interpretation given to words (boundary indicated by the zigzags) in one language may overlap that given in other languages. Adapted from DiMarco, Hirst, and Stede.[359]

overload, *spill* and abbreviations such as *fft*, *cse*, *sql*), or education of the reader (e.g., familiarity with mathematics to a certain level).

- *Readers* of the source code will make assumptions about the intent of the original author. An explicit assumption might be *trusting* the original author (e.g., "the software has been working well"). The reader may have limited time available and trust may seem like the best way of reducing the time they need to invest; whether this trust is justified is not the issue here. The implicit assumptions might include: the reader and author sharing common ground, that the original author had intentions about the identifiers used in a particular section of source (original authors are often seen as mythological figures; it is possible that no particular interpretation was intended), and degree of familiarity with the application domain.

Given the many different kinds of assumptions that different developers may make, people might wonder how readers can be expected to obtain helpful semantic information from identifier spellings. The answer is that often they aren't. There are two routes by which different people might arrive at similar semantic interpretations for an identifier's spelling:

1. *Shared knowledge.* Shared knowledge may occur through culture, natural language usage, or specific learning experiences. An example of where differences in shared knowledge produce differences in performance is the treatment of time in English and Mandarin Chinese. English predominantly treats time as if it were horizontal (e.g., "ahead of time", "push back the deadline"), while Mandarin often, but not always, treats it as being vertical (an English example of vertical treatment is "passes down the generations").[1208] A study by Boroditsky[140] found that English subjects responded more quickly (by approximately 10%) to a question about dates (e.g., "Does March come before April?") if the previous question they had answered had involved a horizontal scenario (e.g., "X is ahead of Y") than if the previous question had involved a vertical scenario (e.g., "X is below Y"). These results were reversed when the subjects were Mandarin speakers and the questions were in Mandarin.
2. *Shared behavior.* Possible shared behaviors include effort minimization, universal grammar, and universal category formation. An example of how shared behavior can affect people's communication was shown by the results of a study by Beun and Cremers.[118] They gave pairs of subjects, a builder and an instructor, the task of building a replica of a building that was only visible to the instructor. The two subjects were seated at a table and could speak to each other and could see each others' hands, but there was no other mode of communication. A pile of blocks of different colors, shapes, and sizes, only visible to the builder, were provided. The block building had to be done on a plate visible to both subjects. Both subjects spoken conversation and hand gestures were recorded. The behavior predicted by Beun and Cremers is based on the *principle of minimal cooperative effort*, where the speaker and

addressee not only try to say as little as possible together, they also try to do as little as possible. For instance, what features of an object should be used in a description? This principle suggests that people will prefer absolute (e.g., *black* or *square*) rather than relative (e.g., *darkest*, *longest*) features. This is because absolute features only require one object to be taken into account, while relative features requires comparison against other objects. The results found that 63% of referential acts used absolute features only, 19% used a combination of absolute and relative features, and 1% used relative features only. A pointing action, with the hands, was used in 18% of cases. The study also found evidence for several other hypotheses derived from this principle, including: (1) if the target object is inherently salient within the domain of conversation, reduced information is used; and (2) if the target object is located in the current focus area, only information that distinguishes the object from other objects in the focus area is used.

6.6 Abbreviating

abbreviating identifier

Developers sometimes form an identifier by abbreviating the words of a descriptive phrase or sentence (e.g., `first_elem_ptr` might be derived from *a pointer to the first element of an array*). While the words in a phrase or complete sentence are rarely abbreviated in everyday life, continued use may result in a multiword phrase eventually being replaced by an acronym (e.g., *Light Amplification by Stimulated Emission of Radiation* became *LASER* and eventually *laser*). For this reason, there does not appear to be any published research on the creation of abbreviations from multi-word phrases.

This subsection discusses the issues associated with shortening the words (it is not unknown for single words to be shortened) used to create an identifier (the issue of source filename abbreviations is discussed elsewhere). The following are some of the issues associated with using shortened forms:

file name 108 abbreviations

- Shortening is likely to change the amount of cognitive resources needed by readers of the source to process an identifier containing them. This is because abbreviation changes the distribution of character sequences, with infrequent or never-seen character pairs occurring more often. It may also remove any obvious grapheme-to-phoneme mapping, making it harder to create a pronunciation

word 792 pronounceability

- Fewer characters means less effort is needed to type the character sequence denoting an identifier
- Reducing the number of characters in an identifier can simplify the visual organization of source code (i.e., by removing the need needing to split an expression or statement over more than one line)
- While some abbreviations may have semantic associations for the original developer, these are often not understood or are forgotten by subsequent readers of the source. Such identifier spellings are then treated as a random sequence of characters

Word shortening can be studied by asking people to create shortened forms of words and phrases;[579, 1308] by analyzing the shortened forms occurring in prose,[1273] source code,[796] speech,[204] or from a purely information content point of view.[145] These studies investigated the form of the abbreviations created, not the circumstance under which people decide to create or use an abbreviation. A number of commonly occurring patterns to the shortened forms have been found, including:

- Vowel deletion— sometimes known as *contraction* (e.g., *search*⇒`srch` and *pointer*⇒`pntr`)
- Truncation of trailing characters (e.g., *audit* ⇒ `aud` and *catalog* ⇒ `cat`)
- A combination of vowel removal and truncation (e.g., *pointer*⇒`ptr` and *temporary*⇒`tmp`)
- Using the first letter of each word (e.g., *customer query number*⇒`cqn` and *first in first out* ⇒ `fifo`)
- Phonetic abbreviations— simply changing one grapheme to another that represents the same phoneme (e.g., *ph*⇒*f*) or digits may be used (e.g., *straight*⇒*str8*)

These abbreviations may in turn be concatenated together to create specialized instances (e.g., `tmp_cqn`, `cat_pntr`, and `srch4cqn`).

An abbreviation is generally a meaningless sequence of characters unless readers can decode it to obtain the original word. A study by Ehrenreich and Porcu[379] found that readers' performance in reconstructing the original word was significantly better when they knew the rules used to create the abbreviation (81–92% correct), compared to when the abbreviation's rules were not known (at best 62% after six exposures to the letter sequences). In practice imposing a fixed set of word abbreviation rules on the world's software developers is not realistic. Imposing a fixed set of abbreviation rules on the developers within a single development group is a realistic possibility, given automated checking to ensure conformance. However, calculating the likely cost/benefit of imposing such a set of rules is very difficult and these coding guidelines do not discuss the issue further.

The widespread use of phonetic abbreviations is relatively new. The growth in online chat forums and the use of text messaging via mobile phones has significantly increased the number of people who use and understand their use. The extent to which this user population intersects the set of creators of identifier spellings also continues to grow. Some people find this use of abbreviations irritating and irresponsible. These coding guidelines take the position that all character sequences should be judged on their ability to communicate information to the reader. Your author has not been able to find any published studies of the use of phonetic abbreviations and they are not discussed further here.

A study by Frost[456] investigated subjects' performance in reading words with missing vowels. The subjects were experienced readers of Hebrew, a language in which words are usually written with the vowels omitted. The time taken to name words was found to vary linearly with the number of vowels omitted. Missing vowels had no effect on lexical decision performance. One point to note is that there was only one word corresponding to each letter sequence used in the study. It is not known how reader performance would vary if there was more than one word matching a vowel-free written form.

The results of two studies asking people to abbreviate the words and phrases they were given showed a number of common points. The common methods used to create abbreviations were the first four of those listed above. Even the shortest words usually had more than one abbreviation (mean of 3.35 in Hodge et al. and 5.73 in Streeter et al), with the average number of abbreviations per word increasing with word length (mean of 6.0 in Hodge et al. and 18.0 in Streeter et al). The most common algorithm used for shorter words was vowel deletion, while longer words tended to be truncated. Streeter et al point out that vowel deletion requires producing the whole word and then deleting the vowels, an effort-prone and time-consuming task for long words (small words probably being handled as a single *chunk*). In the case of polysyllabic words truncation produces short abbreviations, which are easy to produce and only require a single change to the word output strategy (cut it short).

- A study by Hodge and Pennington[579] asked subjects to create a personal (one they might use for their own private writing) and a general (one that could be understood by other people) abbreviation from words containing between four and nine letters. The results for the personal abbreviations paralleled those of the general abbreviations. Male subjects made greater use of vowel removal for the longer words than female subjects (who preferred truncation for longer words). The percentage of the original word's letters used in the abbreviation decreased with word length (from 70–75% for shorter words to 58–65% for longer words). The abbreviations of more frequent words contained fewer letters than less frequent words. A separate group of subjects were given the abbreviations created by the first group and asked to reconstruct the original words. The mean reconstruction rate at all word lengths was 67%.

  792 word frequency

- A study by Streeter, Ackroff, and Taylor[1308] investigated the rules used by people to create abbreviations. Subjects were asked to produce "good abbreviations" for 81 computer command names and arguments (e.g., *move* and "usage billing number"). Analysis of the results was based on the number of syllables in a word (between one and four) or the input containing multiple words. The resulting abbreviations were compared against those produced by a variety of algorithms. The performance of the different algorithms varied with the number of syllables (see Figure 792.33).

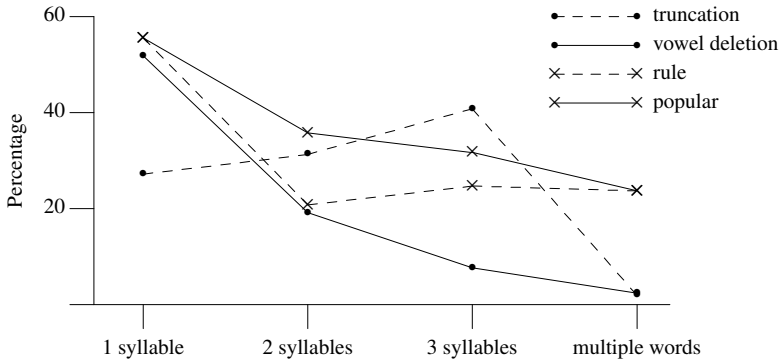


Figure 792.33: Percentage of abbreviations generated using each algorithm. The *rule* case was a set of syllable-based rules created by Streeter et al.; the *popular* case was the percentage occurrence of the most popular abbreviation. Based on Streeter, Ackroff, and Taylor.[1308]

A second group of subjects were asked to learn word/abbreviation pairs. The abbreviations used were those generated by the first group of subjects. The abbreviations used were either the most popular one chosen for a word, or the one obtainable by following the Streeter et al. rules. When given the word and asked for the abbreviation, mean recall performance was 54% correct for the popular abbreviations and 70% for the rule abbreviations (recall rate decreased in both cases as the number of syllables increased, although both were more than 90% for multiple words). Another two experiments, using randomly chosen English words, paralleled the procedure in the first two experiments. However, after learning word/abbreviation pairs, subjects were asked to recall the word when given the abbreviation (mean recall performance was 62.6% correct for the popular abbreviations and 46.7% for the rule abbreviations, with recall rate slowly decreasing in both cases as the number of syllables increased).

An alternative approach to predicting use of abbreviation strategies was studied by Carter and Clopper.[204] Words are abbreviated in both spoken and written forms— for instance, *rhinoceros* ⇒ *rhino* and *telephone* ⇒ *phone*. Subjects were asked to listen to a series of words. After each word, they had to speak the word and then produce a *reduced* spoken form (they were reminded that most of the words would not normally be reduced in everyday speech).

Table 792.21: The syllable most likely to be omitted in a word (indicated by the × symbol) based on the number of syllables (*syl*) and the position of the primary, (*pri*) stressed syllable. Adapted from Carter and Clopper.[204]

Syllables in Word and Primary Stress Position	Syllable(s) 1	Omitted 2	Most 3	Often 4
2syl–1pri		×	–	–
2syl–2pri	×		–	–
3syl–1pri		×	×	–
3syl–2pri	×			–
3syl–3pri		×	×	–
4syl–1pri		×		
4syl–2pri			×	×
4syl–3pri	×	×		×

Carter and Clopper drew three conclusions from the results (see Table 792.21):

1. the stressed syllable is nearly always preserved,
2. the initial syllable is preserved more often than omitted, and
3. only when the final syllable of a two syllable word contains the stress is that syllable preserved more often than it is omitted.

A study by Bourne and Ford[145] investigated word abbreviation from an information content point of view. They looked at thirteen different algorithms capable of reducing an arbitrary word to a predefined number of letters. Algorithm quality was measured by the ability to map different words to different abbreviations. The consistently best algorithm dropped every second letter (this rule was applied iteratively on the successively shorter letter sequences until the desired number of characters was obtained) and appended a check letter (algorithmically derived from the discarded letters). While this algorithm might be of interest when automatically generating identifier spellings, we are only interested in human-created spellings here.

The following studies investigated the interpretation of existing abbreviations in various contexts:

- Sproat, Black, Chen, Kumar, Ostendorf, and Richards[1273] give a detailed discussion of the issues involved in converting what they call *nonstandard* words to spoken words (in the context of text to speech synthesis). What appears to be an abbreviation may actually be a nonstandard word. For instance, `Article_IV` is probably pronounced "article four", `Henry_IV` is pronounced as "Henry the fourth", while `IV_drip` is probably pronounced "I V drip".
- Laitinen, Taramaa, Heikkilä, and Rowe[796] built a tool, `InName`, to *disabbreviate* source code. `InName` used a simple grammar to describe the components of an identifier spelling. This broke the spelling into short letter sequences which were assumed to be either abbreviations of words or words (e.g., `boOffsetMeasDone` ⇒ `bo`, `Offset`, `Meas`, `Done`). A relatively small dictionary of around 1,000 entries was used to detect words, plus a list of 300 common abbreviations (e.g., len ⇒ length, curr ⇒ current). A GUI interface highlighted an abbreviated name and listed possible nonabbreviated forms (e.g., `tmpnamelen` ⇒ `temporary_name_length`). The user could accept one of the suggested forms or type in their own choice. The results of not abbreviating five applications are shown in Table 792.22.

Table 792.22: Five different applications (A–E) unabbreviated using `InName`, by five different people. Application C had many short names of the form `i`, `m`, `k`, and `r2`. Adapted from Laitinen.[796]

Application	A	B	C	D	E
Source lines	12,075	6,114	3,874	6,420	3,331
Total names	1,410	927	439	740	272
Already acceptable	5.6	3.1	8.7	9.3	11.0
Tool suggestion used	42.6	44.7	35.3	46.8	41.5
User suggestion used	39.6	29.3	15.0	30.7	43.8
Skipped or unknown names	12.2	22.9	41.0	13.2	3.7
User time (hours)	11	5	4	4	3

- A study by Cutler and Carter[304] found that 85% of English lexical words (i.e., excluding function words) begin with a strong syllable. They proposed that such a strategy simplified the problem listeners faced in identifying the start of words in continuous speech.
- A study by Anquetil and Lethbridge[48] investigated the abbreviations used to name files (this is discussed elsewhere). 108 file name abbreviations

6.7 Implementation and maintenance costs

Encoding information in the sequence of characters forming an identifier's spelling sounds attractive and this usage often occurs in automatically generated code. However, human-written code is invariably maintained by humans and a number of possible human-related factors ought to be taken into account. While it might not yet be possible to obtain reliable figures on these costs, the main ones are listed here for future reference, including:

- Keeping the information up-to-date during source code maintenance and updates.
- The cost of new readers learning to interpret the encoding used.

- The probability of correctly interpreting the character sequences used. For instance, a numeric value may indicate some maximum value, but this maximum may be the largest representable value (a representation attribute) or the largest value an object is expected to hold (an application attribute).
- The cost of processing the character encoding when reading source. (It took developers many years of practice to achieve fluency in reading text in their native language and they are likely to require practice at decoding identifier spellings before they can read them as fluently.)

The only information likely to be needed every time the identifier is read is the semantics of what it denotes. Experience suggests that this information rarely changes during the development and maintenance of a program.

6.8 Typing mistakes

typing mistakes

A summary of the results of typing studies would include two factors of relevance to this identifier guidelines section. Hitting a key adjacent to the correct one is the largest single contributor (around 35%) to the number of typing mistakes made by people. Performance is affected by the characteristics of the typists native written language (word frequency and morphology).

morphology 792

Researchers studying typing often use skilled typists as subjects (and build models that mimic such people[1188]). These subjects are usually asked to make a typewritten copy of various forms of prose— the kind of task frequently performed by professional typists, the time taken and errors made being measured. Software developers are rarely skilled typists and rarely copy from written material. (It is often created in the developer's head on the fly, and a theory of developer typing performance would probably need to consider these two processes separately.)

These coding guidelines assume that developers' typing mistakes will follow the same pattern as those of typists, although the level of performance may be lower. It is also assumed that the primary input device will be a regular-size keyboard and not one of those found on mobile computers.[882]

- A study by Shaffer and Hardwick[1216] asked qualified touch typists to type text, the characteristics of which varied. The five different kinds of text were: Prose, an article on gardening; Word, a random arrangement of the words in the previous article; Syllable, obtained by shuffling the spaces between the words of the article into syllable boundaries within the words; First-order, random letter strings having the same distribution of letters as the article; and Zero-order, random letter strings with all letters being equally probable.

Table 792.23: Distribution of mistakes for each kind of text. Unparenthesized values are for subjects who made fewer than 2.5% mistakes, and parenthesized values for subjects who made 2.5% or more mistakes. Omission— failing to type a letter; response— hitting a key adjacent to the correct one; reading— mistakes were those letters that are confusable visually or acoustically; context — transpositions of adjacent letters and displacements of letters appearing within a range of three letters left or right of the mistake position; random— everything else. When a mistake could be assigned to more than one category, the category appearing nearer the top of the table was chosen. Adapted from Shaffer.[1216]

Kind of mistake	Prose	Word	Syllable	First Order	Zero Order	Total
Omission	19 (21)	11 (23)	24 (36)	15 (46)	34 (82)	103 (208)
Response	19 (25)	31 (38)	27 (53)	32 (43)	108 (113)	217 (272)
Reading	3 (2)	2 (0)	8 (15)	14 (20)	20 (41)	47 (78)
Context	19 (27)	19 (17)	34 (30)	56 (51)	46 (40)	174 (165)
Random	3 (5)	2 (6)	4 (11)	13 (15)	22 (41)	44 (78)
Total	63 (80)	65 (84)	97 (145)	130 (175)	230 (317)	585 (801)

The results (see Table 792.23) show that hitting a key adjacent to the correct one was the largest single contributor (around 35%) to the number of mistakes. Surprisingly, both the number of mistakes and typing rate were the same for prose and random word ordering. Text containing words created using purely random letter sequences had the highest rate of typing mistakes (and the slowest typing

rate), almost twice that of text created using the distribution of letters found in English. Shaffer and Hardwick performed a second experiment to investigate the reasons for the significant difference in typing performance when first- or zero-order words were used. Was it caused by a decline in syllable-like sequences in the words because of fewer vowels, or because of an increase in the less frequently used letters of the alphabet? Every letter of the alphabet was used 10 times to create passages of 52 five-letter words and 16 fifteen-letter words (plus a single 20-letter word). For one set of passages the letters in each word were randomly selected; in the other set an attempt was made to create words containing readable syllables (e.g., *spowd*, *throx*).

Table 792.24: Mean response time per letter (in milliseconds). Right half of the table shows mean response times for the same subjects with comparable passages in the first experiment. Adapted from Shaffer.[1216]

	Syllable	Random		First Order	Zero Order
5-letter	246	326	Fixed	236	344
15-letter	292	373	Random	242	343

The only results reported (see Table 792.24) were response times for letters typed, not the number of mistakes. These results show that performance for text containing words having readable syllables was significantly better than words having a random sequence of letters. Since both passages contained the same number of occurrences of each letter, the difference was not caused by a decrease in the number of vowels or an increase in the number of infrequently used letters. Performance was slower for the passage containing longer words.

- A study by Gentner, Larochelle, and Grudin[481] also found that rate of typing was affected by letter digraph frequency and word frequency in the typist's natural language (they did not measure error mistake rates). The position of the digraph within the word and syllable boundaries had a smaller affect on performance.
- A study by Schoonard and Boies[1203] taught subjects to use abbreviations for commonly occurring words (so-called *short-type*; the intent being to increase typing performance by having the word processor used automatically expand the abbreviations). The results showed an average short-type detection rate of 93.2% (of those possible) and that typing rate (in characters per second) was not affected by use of short-type (error rates were only given for short-type). Developers often abbreviate words when creating identifier names (but editors rarely expand them).

0 people
error rates
792 abbreviating
identifier

- Studies have analyzed the mistakes made by each finger of each hand. Software developers rarely touch type, often using a few fingers from each hand. While *Stewardesses* may be the longest English word touch typed with the left hand only, most developers are likely to use fingers from the right hand. For this reason these studies are not considered applicable here.

6.9 Usability of identifier spelling recommendations

The computational resources needed to rapidly check a character sequence against a list of several million character sequences is well within the capabilities of computers used for software development today. However, the cost of maintaining an up to-date list of identifiers currently used within a software product under active development can be a non-trivial task and may result in developers having to wait for a relatively long period of time for a proposed choice of identifier spelling to be checked against existing identifier spellings. One way of reducing the identifier database maintenance resources required is to reduce the number of identifiers that need to be checked. All identifiers have attributes other than their spelling (they all exist in some name space and scope, have a linkage and some have a type), and it might be possible to take advantage of the consequences of an identifier having these attributes; for instance:

identifier
guideline significant characters

438 name space
400 scope
420 linkage
472 types

- If the identifier `X_1` is not visible at the point in the source where a developer declares and uses the identifier `X_2`, a reference to `X_2` mistyped as `X_1` will result in a translator diagnostic being issued.

400 visible
identifier

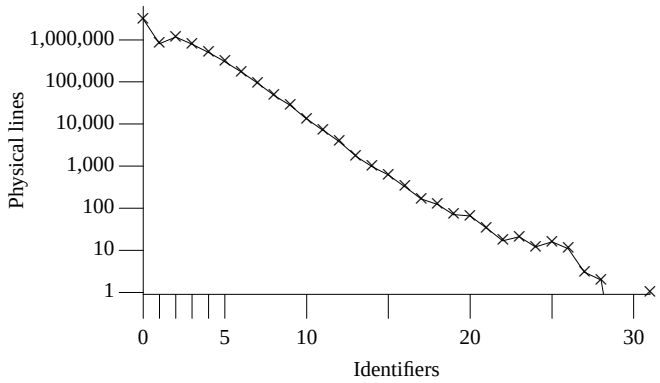


Figure 792.34: Number of physical lines containing a given number of identifiers. Based on the visible form of the `.c` files.

name space 438

- If the identifiers `X_1` and `X_2` are in different name spaces, a mistyped reference to one can never result in a reference to the other.
- If the identifiers `X_1` and `X_2` are objects or functions having incompatible types, a mistyped reference to one, which refers to the other, is likely to result in a translator diagnostic being issued.

Mistyping an identifier only becomes a fault if the usage does not cause a diagnostic to be issued by a translator. (A developer who visually confuses two identifiers can create a fault that does not generate a translator diagnostic by writing code that makes use of incorrect identifier information without directly referencing the confused identifiers.)

While modifications to existing source may not result in new identifiers being declared, it is possible for the C language attributes of an existing identifier to be changed. Experience suggests that modifications rarely change an identifier's name space, but that changes of scope or linkage (which controls visibility) is relatively common; for instance, moving the declaration of an identifier from block scope to file scope or changing its linkage from internal to external (experience suggests that these changes rarely occur in the other direction). Changes to the type of an object might include a change of integer type or new members being added to the structure type it has.

scope 410
overlapping

This usage pattern suggests the following deviation:

Dev 792.3

Every identifier need only be compared against every other identifier in the same name space in the visible source of a program.

Semantics

An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. 793

Commentary

A restatement of information given in the Syntax clause.

C90

Explicit support for other characters is new in C99.

Lowercase and uppercase letters are distinct. 794

Commentary

The reason the standard needs to explicitly specify that lowercase and uppercase letters are distinct is because in many computer languages they are not. In some tokens lowercase and uppercase letters may not be treated as being distinct.

1911 header name significant characters

Other Languages

Some languages (e.g., Ada, Fortran, Lisp, and Pascal) support the use of lowercase letters in an identifier, but treat them as being equivalent to their corresponding uppercase letters. Java and Modula-2 treat lower- and uppercase letters as being distinct.

Coding Guidelines

The visual similarity of these letters is discussed elsewhere.

792 character visual similarity

---

795 There is no specific limit on the maximum length of an identifier.

Commentary

The standard does specify a minimum limit on the number of characters a translator must consider as significant. Implementations are free to ignore characters once this limit is reached. The ignored characters do not form part of another token. It is as if they did not appear in the source at all.

282 internal identifier significant characters
283 external identifier significant characters

C90

The C90 Standard does not explicitly state this fact.

Other Languages

Few languages place limits on the maximum length of an identifier that can appear in a source file. Like C, some specify a lower limit on the number of characters that must be considered significant.

Coding Guidelines

Using a large number of characters in an identifier spelling has many potential benefits; for instance, it provides the opportunity to supply a lot of information to readers, or to reduce dependencies on existing reader knowledge by spelling words in full rather than using abbreviations. There are also potential costs; for instance, they can cause visual layout problems in the source (requiring new-lines within an expression in an attempt to keep the maximum line length within the bounds that can be viewed within a fixed-width window), or increase the cognitive effort needed to visually scan source containing them.

The length of an identifier is not itself directly a coding guideline issue. However, length is indirectly involved in many identifier memorability, confusability, and usability issues, which are discussed elsewhere.

792 identifier syntax

Usage

The distribution of identifier lengths is given in Figure 792.7.

---

796 Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in annex D.[60)]

identifier UCN

Commentary

Using other UCNs results in undefined behavior (in some cases even using these UCNs can be a constraint violation). These character encodings could be thought of as representing letters in the specified national character set.

816 UCNs not basic character set

C90

Support for universal character names is new in C99.

Other Languages

The ISO/IEC 10646 standard is relatively new and languages are only just starting to include support for the characters it specifies. Java specifies a similar list of UCNs.

28 ISO 10646

Common Implementations

A collating sequence may not be defined for these universal character names. In practice a lack of a defined collating sequence is not an implementation problem. Because a translator only ever needs to compare the spelling of one identifier for equality with another identifier, which involves a simple character-by-character comparison (the issue of the ordering of diacritics is handled by not allowing them to occur in an identifier).

Support for this functionality is new and the extent to which implementations are likely to check that UCN values fall within the list given in annex D is not known.

Coding Guidelines

The intended purpose for supporting universal character names in identifiers is to reduce the developer effort needed to comprehend source. Identifiers spelled in the developer's native tongue are more immediately recognizable (because of greater practice with those characters) and also have semantic associations that are more readily brought to mind.

ISO 10646 28 The ISO 10646 Standard does not specify which languages contain the characters it specifies (although it does give names to some sets of characters that correspond to a language that contains them). The written form of some human languages share common characters; for instance, the characters *a* through *z* (and their orthography 792 uppercase forms) appear in many European orthographies. The following discussion refers to using UCNs from more than one human language. This is to be taken to mean using UCNs that are not part of the written form of the native language of the developer (the case of developers having more than one native language is not considered). For instance, the character *a* is used in both Swedish and German; the character û is used in Swedish, but not German; the character ß is used in German but not Swedish. Both Swedish and German developers would be familiar with the character *a*, but the character ß would be considered foreign to a Swedish developer, and the character û foreign to the German.

Some coding guideline documents recommend against the use of UCNs. Their use within identifiers can increase the portability cost of the source. The use of UCNs is an economic issue; the potential cost of not permitting their use in identifiers needs to be compared against the potential portability benefits. (Alternatively, the benefits of using UCNs could be compared against the possible portability costs.)

Given the purpose of using UCNs, is there any rationale for identifiers to contain characters from more than one human language? As an English speaker, your author can imagine a developer wanting to use an English word, or its common abbreviation, as a prefix or suffix to an identifier name. Perhaps an Urdu speaker can imagine a similar usage with Urdu words. The issue is whether the use of characters in the same identifier from different human languages has meaning to the developers who write and maintain the source.

Identifiers very rarely occur in isolation. Should all the identifiers in the same function, or even source file, only contain UCNs that form the set of characters used by a single human language? Using characters from different human languages when it is possible to use only characters from a single language, potentially increases the cost of maintenance. Future maintainers are either going to have to be familiar with the orthography and semantics of the two human languages used or spend additional time processing instances of identifiers containing characters they are not familiar with. However, in some cases it might not be possible to enforce a *single human language* rule. For instance, a third-party library may contain callable functions whose spellings use characters from a human language different from that used in the source code that contains calls to it.

Support for the use of UCNs in identifiers is new in C99 (and other computer languages) and at the time of this writing there is almost no practical experience available on the sort of mistakes that developers make with them.

The initial character shall not be a universal character name designating a digit. 797

Commentary

identifier 792 syntax The terminal *identifier-nondigit* that appears in the syntax implies that the possible UCNs exclude the digit characters. Also the list given in annex D does not include the digit characters. This means that an identifier containing a UCN designating a digit in any position results in undefined behavior.

The syntax for constants does not support the use of UCNs. This sentence, in the standard, reminds implementors that such usage could be supported in the future and that, while they may support UCN digits within an identifier, it would not be a good idea to support them as the initial character. 822 constant syntax

Table 797.1: The Unicode digit encodings.

Encoding Range	Language	Encoding Range	Language
0030–0039	ISO Latin-1	0BE7–0BEF	Tamil (has no zero)
0660–0669	Arabic–Indic	0C66–0C6F	Telugu
06F0–06F9	Eastern Arabic–Indic	0CE6–0CEF	Kannada
0966–096F	Devanagari	0D66–0D6F	Malayalam
09E6–09EF	Bengali	0E50–0E59	Thai
0A66–0A6F	Gurmukhi	0ED0–0ED9	Lao
0AE6–0AEF	Gujarati	FF10–FF19	Fullwidth
0B66–0B6F	Oriya digits		

C++

This requirement is implied by the terminal non-name used in the C++ syntax. Annex E of the C++ Standard does not list any UCN digits in the list of supported UCN encodings.

Other Languages

Java has a similar requirement.

Coding Guidelines

The extent to which different cultural conventions support the use of a digit as the first character in an identifier is not known to your author. At some future date the Committee may chose to support the writing of integer constants using UCNs. If this happens, any identifiers that start with a UCN designating a digit are liable to result in syntax violations. There does not appear to be a worthwhile benefit in a guideline recommendation dealing with the case of an identifier beginning with a UCN designating a digit.

Example

```
1  int \u1f00\u0ae6;
2  int \u0ae6;
```

---

798 An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; identifier multibyte character in

Commentary

Prior to C99 there was no standardized method of representing nonbasic source character set characters in the source code. Support for multibyte characters in string literals and constants was specified in C90; some implementations extended this usage to cover identifiers. They are now officially sanctioned to do this. Support for the ISO 10646 Standard is new in C99. However, there are a number of existing implementations that use a multibyte encoding scheme and this usage is likely to continue for many years. The C committee recognized the importance of this usage and do not force developers to go down a UCN-only path. 28 ISO 10646

The standard says nothing about the behavior of the **__func__** reserved identifier in the case when a function name is spelled using wide characters. 810 __func__

C90

This permission is new in C99.

C++

translation phase 1 116

The C++ Standard does not explicitly contain this permission. However, translation phase 1 performs an implementation-defined mapping of the source file characters, and an implementation may choose to support multibyte characters in identifiers via this route.

Other Languages

While other language standards may not mention multibyte characters, the problem they address is faced by implementations of those languages. For this reason, it is to be expected that some implementations of other languages will contain some form of support for multibyte characters.

Coding Guidelines

universal character name syntax 815

UCNs may be the preferred, C Standard way, of representing nonbasic character set characters in identifiers. However, developers are at the mercy of editor support for how they enter and view characters that are not in the basic source character set.

---

which characters and their correspondence to universal character names is implementation-defined. 799

Commentary

Various national bodies have defined standards for representing their national character sets in computer files. While ISO 10646 is intended to provide a unified standard for all characters, it may be some time before existing software is converted to use it.

ISO 10646 28

Common Implementations

It is common to find translators aimed at the Japanese market supporting JIS, shift-JIS, and EUC encodings (see Table 243.3). These encoding use different numeric values than those given in ISO 10646 to represent the same national character.

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword. 800

Commentary

The Committee could have created a separate name space for keywords and allowed developers to define identifiers having the same spelling as a keyword. The complexity added to a translator by such a specification would be significant (based on implementation experience for languages that support this functionality), while a developer's inability to define identifiers having these spellings was considered a relatively small inconvenience.

C90

This wording is a simplification of the convoluted logic needed in the C90 Standard to deduce from a constraint what C99 now says in semantics. The removal of this C90 constraint is not a change of behavior, since it was not possible to write a program that violated it.

C90 6.1.2

Constraints

In translation phase 7 and 8, an identifier shall not consist of the same sequence of characters as a keyword.

Other Languages

Some languages allow keywords to be used as variable names (e.g., PL/1), using the context to disambiguate intended use.

---

footnote 60

60) On systems in which linkers cannot accept extended characters, an encoding of the universal character name may be used in forming valid external identifiers. 801

Commentary

This is really an implementation tip for translators. The standard defines behavior in terms of an abstract machine that produces external output. The tip given in this footnote does not affect the conformance status of an implementation that chooses to implement this functionality in another way. The only time such a mapping might be visible is through the use of a symbolic execution-time debugging tool, or by having to link against object files created by other translators.

C90

Extended characters were not available in C90, so the suggestion in this footnote does not apply.

215 extended characters

Other Languages

Issues involving third-party linkers are common to most language implementations that compile to machine code. Some languages, for instance Java, define the characteristics of an implementation at translation and execution time. The Java language specification goes to the extreme (compared to other languages) of specifying the format of the generated file object code file.

Common Implementations

There is a long-standing convention of prefixing externally visible identifier names with an underscore character when information on them is written out to an object file. There is little experience available on implementation issues involving UCNs, but many existing linkers do assume that identifiers are encoded using 8-bit characters.

Coding Guidelines

The encoding of external identifiers only needs to be considered when interfacing to, or from code written in another language. Cross-language interfacing is outside the scope of these coding guidelines.

---

802 For example, some otherwise unused character or sequence of characters may be used to encode the **\u** in a universal character name.

Commentary

Some linkers may not support an occurrence of the backslash (\) character in an identifier name. One solution to this problem is to create names that cannot be declared in the source code by the developer; for instance, by deleting the \ characters and prefixing the name with a digit character.

Common Implementations

There are no standards for encoding of universal character names in object files. The requirement to support this form of encoding is too new for it to be possible to say anything about common encodings.

---

803 Extended characters may produce a long external identifier.

Commentary

Here the word *long* does not have any special meaning. It simply suggests an identifier containing many characters.

282 internal identifier significant characters

Implementation limits

---

804 As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier;

Implementation limits

Commentary

This subclause lists a number of minimum translation limits

276 translation limits

C90

The C90 Standard does not contain this observation.

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

Annex B contains an informative list of possible implementation limits. However, " . . . these quantities are only guidelines and do not determine compliance.".

---

the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). 805

Commentary

external 283 identifier significant characters

External identifiers have to be processed by a linker, which may not be under the control of a vendor's C implementations. In theory, any tool that performs the linking process falls within the remit of the C Committee. However, the Committee recognized that, in practice, it is not always possible for translator vendors to supply their own linker. The limitations of existing linkers needed to be factored into the limits specified in the standard.

internal 282 identifier significant characters

Internal identifiers only need to be processed by the translator and the standard is in a strong position to specify the behavior.

Other Languages

Most other language implementations face similar problems with linkers as C does. However, not all language specifications explicitly deal with the issue (by specifying the behavior). The Java standard defines a complete environment that handles all external linkages.

Coding Guidelines

What are the costs associated with a change to the linkage of an identifier during program maintenance, from internal linkage to external linkage? (Experience shows that identifier linkage is rarely changed from external to internal?)

external 283 identifier significant characters identifier 792 number of characters

In most cases implementations support a sufficiently large number of significant characters in an external name that a change of identifier linkage makes no difference to its significant characters (i.e., the number of characters it contains falls inside the implementation limit). In those cases where a change of identifier linkage results in some of its significant characters being ignored, the affect may be benign (there is no other identifier defined with external linkage whose name is the same as the truncated name) or results in undefined behavior (the program defines two identifiers with external linkage with the same name).

external 1818 linkage exactly one external definition

The number of significant characters in an identifier is implementation-defined. 806

Commentary

internal 282 identifier significant characters

Subject to the minimum requirements specified in the standard.

C++

2.10p1 *All characters are significant.*[20)]

References to the same C identifier, which differs after the last significant character, will cause a diagnostic to be generated by a C++ translator.

There is also an informative annex which states:

Annex Bp2

Number of initial characters in an internal identifier or a macro name [1024]

Number of initial characters in an external identifier [1024]

Other Languages

Some languages require all characters in an identifier to be significant (e.g., Java, Snobol 4), while others don't (e.g., Cobol, Fortran).

Common Implementations

It is rare to find an implementation that does not meet the minimum limits specified in the standard. A few translators treat all identifiers as significant. Most have a limit of between 256 and 2,000 significant characters. The POSIX standard requires that any language that binds to its API needs to support 14 significant characters in an external identifier.

Coding Guidelines

While the C90 minimum limits for the number of significant characters in an identifier might be considered unacceptable by many developers, the C99 limits are sufficiently generous that few developers are likely to complain.

Automatically generated C source sometimes relies on a large number of significant characters in an identifier. This can occur because of the desire to simplify the implementation of the generator. Character sequences in different offsets within an identifier might be reserved for different purposes. Predefined default character sequence is used to pad the identifier spelling where necessary.

As the following example shows, it is possible for a program's behavior to change, both when the number of significant identifiers is increased and when it is decreased.

```
1  /*
2   * Yes, C99 does specify 64 significant characters in an internal
3   * identifier.  But to keep this example within the page width
4   * we have taken some liberties.
5   */
6
7  extern float _________1_________2_________3___bb;
8
9  void f(void)
10 {
11 int _________1_________2_________3___ba;
12
13 /*
14  * If there are 34 significant characters, the following operand
15  * will resolve to the locally declared object.
16  *
17  * If there are 35 significant characters, the following operand
18  * will resolve to the globally declared object.
19  */
20 _________1_________2_________3___bb++;
21 }
22
23 void g(void)
24 {
25 int _________1_________2_________3___aa;
26
27 /*
28  * If there are 34 significant characters, the following operand
29  * will resolve to the globally declared object.
30  *
31  * If there are 33 significant characters, the following operand
32  * will resolve to the locally declared object.
33  */
```

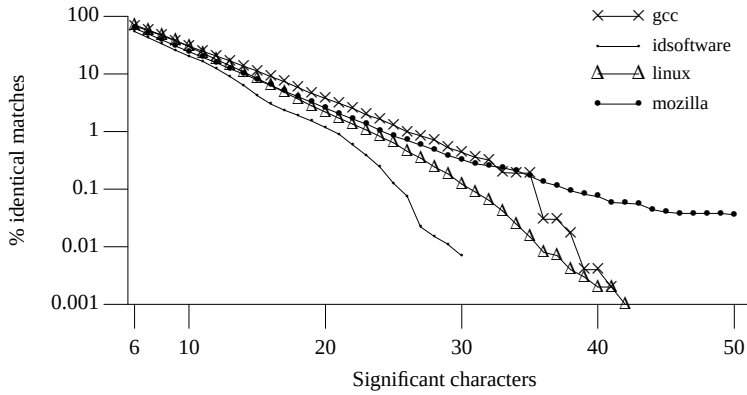


Figure 806.1: Occurrence of unique identifiers whose significant characters match those of a different identifier (as a percentage of all unique identifiers in a program), for various numbers of significant characters. Based on the visible form of the `.c` files.

```
34  __________1_________2_________3___bb++;
35  }
```

The following issues need to be addressed:

- All references to the same identifier should use the same character sequence; that is, all characters are intended to be significant. References to the same identifiers that differ in nonsignificant characters need to be treated as faults.
- Within how many significant characters should different identifiers differ? Should identifiers be required to differ within the minimum number of significant characters specified by the standard, or can a greater number of characters be considered significant?

reading 770 kinds of
identifiers 792 Greek readers
word 770 reading individual

Readers do not always carefully check all characters in the spelling of an identifier. The contribution made by characters occurring in different parts of an identifier will depend on the pattern of eye movements employed by readers, which in turn may be affected by their reasons for reading the source, plus cultural factors (e.g., direction in which they read text in their native language, or the significance of word endings in their native language). Characters occurring at both ends of an identifier are used by readers (at least native English- and French-speaking ones) when quickly scanning text.

Cg 806.1 When performing similarity checks on identifiers, all characters shall be considered significant.

Any identifiers that differ in a significant character are different identifiers. 807

Commentary

In many cases different identifiers also denote different entities. In a some cases they denote the same entity (e.g., two different typedef names that are synonyms for the type **`int`**).

Other Languages

This statement is common to all languages (but it does not always mean that they necessarily denote different entities).

Coding Guidelines

Identifiers that differ in a single significant character may be considered to be

- different identifiers by a translator, but considered to be the same identifier by some readers of the source (because they fail to notice the difference).
- the same identifiers by a translator (because the difference occurs in a nonsignificant character), but considered to be different identifiers by some readers of the source (because they treat all characters as being significant).
- identifiers by both a translator and some readers of the source.

The possible reasons for readers making mistakes are discussed elsewhere, as are the guideline recommendations for reducing the probability that these developer mistakes become program faults.

0 developer errors
792 identifier filtering spellings

Example

```
1  extern int e1;
2  extern long e1;
3  extern int a_longer_more_meaningful_name;
4  extern int a_longer_more_meeningful_name;
5  extern int a_meaningful_more_longer_name;
```

808 If two identifiers differ only in nonsignificant characters, the behavior is undefined.

Commentary

While the obvious implementation strategy is to ignore the nonsignificant characters, the standard does not require implementations to use this strategy. To speed up identifier lookup many implementations use a hashed symbol table— the hash value for each identifier is computed from the sequence of characters it contains. Computing this hash value as the characters are read in, to form an identifier, saves a second pass over those same characters later. If nonsignificant characters were included in the original computed hash value, a subsequent occurrence of that identifier in the source, differing in nonsignificant characters, would result in a different hash value being calculated and a strong likelihood that the hash table lookup would fail.

Developers generally expect implementations to ignore nonsignificant characters. An implementation that behaved differently because identifiers differed in nonsignificant characters might not be regarded as being very user friendly. Highlighting misspellings that occur in nonsignificant characters is not always seen in a positive light by some developers.

C++

In C++ all characters are significant, thus this statement does not apply in C++.

Other Languages

Some languages specify that nonsignificant characters are ignored and have no effect on the program, while others are silent on the subject.

Common Implementations

Most implementations simply ignore nonsignificant characters. They play no part in identifier lookup in symbol tables.

Coding Guidelines

The coding guideline issues relating to the number of characters in an identifier that should be considered significant are discussed elsewhere.

792 identifier guideline significant characters

809 **Forward references:** universal character names (6.4.3), macro replacement (6.10.3).

6.4.2.2 Predefined identifiers

Semantics

__func__ The identifier **__func__** shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration 810

```
static const char __func__[] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.[61)]

Commentary

Implicitly declaring **__func__** immediately after the opening brace in a function definition means that the first, developer-written declaration within that function can access it. Giving **__func__** static storage duration enables its address to be referred to outside the lifetime of the function that contains it (e.g., enabling a call history to be displayed at some later stage of program execution). This is not a storage overhead because space needs to be allocated for the string literal denoted by **__func__**. The **const** qualifier ensures that any attempts to modify the value cause undefined behavior. The identifier **__func__** has an array type, and is not a string literal, so the string concatenation that occurs in translation phase 6 is not applicable.

translation phase 6 135

This identifier is useful for providing execution trace information during program testing. Developers who make use of UCNs may need to ensure that the library they use supports the character output required by them:

```
#include <stdio.h>

void \u30CE(void)
{
printf ("Just entered %s\n", __func__);
}
```

identifier multibyte character in 798

The issue of wide characters in identifiers is discussed elsewhere.

Which function name is used when a function definition contains the **inline** function specifier? In:

```
#include <stdio.h>

inline void f(void)
{
printf("We are in %s\n", __func__);
}

int main(void)
{
f();
printf("We are in %s\n", __func__);
}
```

the name of the function `f` is output, even if that function is inlined into `main`.

C90

Support for the identifier **__func__** is new in C99.

C++

Support for the identifier **__func__** is new in C99 and is not available in the C++ Standard.

Common Implementations

A translator only needs to declare **__func__** if a reference to it occurs within a function. An obvious storage saving optimization is to delay any declaration until such time as it is known to be required. Another optimization is for the storage allocated for **__func__** to exactly overlay that allocated to the string literal. Allocating storage for a string literal and copying the characters to the separately allocated object it initializes is not necessary when that object is defined using the **const** qualifier. `gcc` also supports the built-in form **__FUNCTION__**.

Example

Debugging code in functions can provide useful information. But when there are lots of functions, the quantity of useless information can be overwhelming. Controlling which functions are to output debugging information by using conditional compilation requires that code be edited and the program rebuilt.

The names of functions can be used to dynamically control which functions are to output debugging information. This control not only reduces the amount of information output, but can also reduce execution time by orders of magnitude (output can be a resource-intense operation).

flookup.h

```
typedef struct f__rec {
                      char *func_name;
                      _Bool enabled;
                      struct f__rec *next;
                      } func__list;

extern _Bool func_lookup(func__list *, char *);

/*
 * Use the name of the function to control whether debugging is
 * switched on/off.  lookup is only called the first time this code
 * is executed, thereafter the value f___l->enabled can be used.
 */
#define D_func_trace(func_name, code)  {           \
                 static func__list * f___l = NULL; \
                 if (f___l ? f___l->enabled : lookup(&f___l, func_name)) \
                    {code}                                               \
                 }
```

flookup.c

```
#include <stdbool.h>

#include "flookup.h"

/*
 * A fixed list of functions and their debug mode.
 * We could be more clever and make this a list which
 * could be added to as a program executes.
 */
static struct {
              char *func_name;
              _Bool enabled;
              func__list *traces_seen;
              } lookup_table[] = {
                                  "abc", true, NULL,
                                  NULL, false, NULL
                                  };

_Bool func_lookup(func__list *f_list, char *f_name)
{
/*
 * Loop through lookup_table looking for a match against f_name.
 * If a match is found, add f_list to the traces_seen list and
 * return the value of enabled for that entry.
 */
}

void change_enabled_setting(char *f_name, _Bool new_enabled)
{
/*
 * Loop through lookup_table looking for a match against f_name.
 * If a match is found, loop over its traces_seen list setting
```

```
33    * the enabled flag to new_enabled.
34    *
35    * This function can switch on/off the debugging output from
36    * any registered function.
37    */
38   }
```

This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5. 811

Commentary

translation phase 5 133

Having the name appearing as if in translation phase 5 avoids any potential issues caused by macro names defined with the spelling of keywords or the name **__func__**. It also enables a translator to have an identifier name and type predefined internally, ready to be used when this reserved identifier is encountered. Translation phase 5 is also where characters get converted to their corresponding members in the execution character set, an essential requirement for spelling a function name. In many implementations the function name written to the object file, or program image, is different from the one appearing in the source. This translation phase 5 requirement ensures that it is not any modified name that is used.

program image 141

Example

```
1   #include <stdio.h>
2
3   #define __func__ __CNUF__
4   #define __CNUF__ "g"
5
6   void f(void)
7   {
8   /*
9    * The implicit declaration does not appear until after preprocessing.
10   * So there is no declaration 'static const char __func__[] = "f";'
11   * visible to the preprocessor (which would result in __func__ being
12   * mapped to __CNUF__ and "f" rather than "g" being output).
13   */
14  printf("Name of function is %s\n", __CNUF__);
15  }
```

EXAMPLE Consider the code fragment: 812

```
#include <stdio.h>
void myfunc(void)
{
        printf("s\n", __func__);
        /* ... */
}
```

Each time the function is called, it will print to the standard output stream:
myfunc

Commentary

This assumes that the standard output stream is not closed (in which case the behavior would be undefined).

Forward references: function definitions (6.9.1). 813

814 61) Since the name **__func__** is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name **__func__**, the behavior is undefined. footnote 61

Commentary

The name is reserved because it begins with two underscores. The fact that the standard defines an interpretation for this name in the identifier name space in block scope does not give any license to the developer to use it in other name spaces or at file scope. This name is still reserved for use in other name spaces and scopes.

C90

Names beginning with two underscores were specified as reserved for any use by the C90 Standard. The following program is likely to behave differently when translated and executed by a C99 implementation.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  int __func__ = 1;
6
7  printf("d\n", __func__);
8  }
```

C++

Names beginning with __ are reserved for use by a C++ implementation. This leaves the way open for a C++ implementation to use this name for some purpose.

6.4.3 Universal character names

815 universal character name syntax

universal-character-name:
    \u *hex-quad*
    \U *hex-quad hex-quad*
hex-quad:
    hexadecimal-digit hexadecimal-digit
        hexadecimal-digit hexadecimal-digit

Commentary

It is intended that this syntax notation not be visible to the developer,when reading or writing the source code that contains instances of this construct. A *universal-character-name* aware editor being used to display the ISO 10646 character represented by the numeric value specified by the *hex-quad* sequence value. Without such editor support, the whole rationale for adding these characters to C, allowing developers to read and write identifiers in their own language, is voided.

C90

Support for this syntactic category is new in C99.

Other Languages

Java calls this lexical construct a *UnicodeInputCharacter* (and does not support the \U form, only the \u one).

Coding Guidelines

It is difficult to imagine developers regularly using UCNs with an editor that does not display UCNs in some graphical form. A guideline recommending the use of such an editor would not be telling developers anything they did not already know.

A number of theories about how people recognize words have been proposed. One of the major issues yet 792 Word recognition models of

to be resolved is the extent to which readers make use of *whole word* recognition versus mapping character sequences to sound (phonological coding). Support for UCNs increases the possibility that developers will encounter unfamiliar characters in source code. The issue of developer performance in handling unfamiliar characters is discussed elsewhere.

reading 792 characters unknown to reader

Example

```
#define foo(x)

void f(void)
{
foo("\\u0123") /* Does not contain a UCN. */
foo(\\u0123);  /* Does contain a UCN. */
}
```

Constraints

UCNs not basic character set

A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (`$`), 0040 (`@`), or 0060 (`‘`), nor one in the range D800 through DFFF inclusive.[62)] 816

Commentary

ISO 10646 28

The ISO 10646 Standard defines the ranges 00 through 01F, and 07F through 09F, as the 8-bit control codes (what it calls *C0* and *C1*). Most of the UCNs with values less than 00A0 represent characters in the basic source character set. The exceptions listed enumerate characters that are in the Ascii character set, but not in the basic source character set. The ranges 0D800 through DBFF and 0DC00 through 0DFFF are known as the surrogate ranges. The purpose of these ranges is to allow representation of rare characters in future versions of the Unicode standard.

This constraint means that source files cannot contain the UCN equivalent for any members of the basic source character set.

Rationale

UCNs are not permitted to designate characters from the basic source character set in order to permit fast compilation times for C programs. For some real world programs, compilers spend a significant amount of time merely scanning for the characters that end a quoted string, or end a comment, or end some other token. Although, it is trivial for such loops in a compiler to be able to recognize UCNs, this can result in a surprising amount of overhead.

A UCN is constrained not to specify a character short identifier in the range 0000 through 0020 or 007F through 009F inclusive for the same reason: this avoids allowing a UCN to designate the newline character. Since different implementations use different control characters or sequences of control characters to represent newline, UCNs are prohibited from representing any control character.

C++

2.2p2 *If the hexadecimal value for a universal character name is less than 0x20 or in the range 0x7F–0x9F (inclusive), or if the universal character name designates a character in the basic source character set, then the program is ill-formed.*

The range of hexadecimal values that are not permitted in C++ is a subset of those that are not permitted in C. This means that source which has been accepted by a conforming C translator will also be accepted by a conforming C++ translator, but not the other way around.

Other Languages

Java has no such restrictions on the hexadecimal values.

Common Implementations

Support for UCNs is new in C99. It remains to be seen whether translator vendors decide to support any UCN hexadecimal value as an extension.

Example

```
1  \u0069\u006E\u0074 glob; /* Constraint violation. */
```

Description

817 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Commentary

UCNs may also appear in comments. However, comments do not have a lexical structure to them. Inside a comment character, sequences starting with \u are not treated as UCNs by a translator, although other tools may choose to do so, in this context. The mapping of UCNs in character constants and string literals to the execution character set occurs in translation phase 5.

The constraint on the range of values that a UCN may take prevents them from being used to represent keywords.

816 UCNs not basic character set

C++

The C++ Standard also supports the use of universal character names in these contexts, but does not say in words what it specifies in the syntax (although 2.2p2 comes close for identifiers).

Other Languages

In Java, *UnicodeInputCharacters* can represent any character and is mapped in lexical translation step 1. It is possible for every character in the source to appear in this form. The mapping only occurs once, so \u005cu005a becomes \u005a, not Z (005c is the Unicode value for \ and 005a is the Unicode character for Z).

Coding Guidelines

UCNs in character constants and string literals are used to represent characters that are output when a program is executed, or in identifiers to provide more readable source code. In the former case it is possible that UCNs from different natural languages will need to be represented. In the latter case it might be surprising if source code contained UCNs from different languages. This usage is a complex one involving issues outside of these coding guidelines (e.g., configuration management and customer requirements) and your author has insufficient experience to know whether any guideline recommendations might be worthwhile.

Some of the coding guideline issues relating to the use of characters outside of the basic execution character set are discussed elsewhere.

238 multibyte character source contain

Example

```
1  #include <wchar.h>
2
3  int \u0386\u0401;
4  wchar_t *hello = "\u05B0\u0901";
```

Semantics

short identifier

The universal character name \U*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*.[63)] 818

Commentary

The standard specifies how UCNs are represented in source code. A development environment may chose to provide, to developers, a visible representation of the UCN that matches the glyph with the corresponding numeric value in ISO 10646. The ISO 10646 BNF syntax for short identifiers is:

ISO 10646 short identifier

```
{ U | u } [ {+}(xxxx | xxxxx | xxxxxx) | {-}xxxxxxxx ]
```

where x represents a hexadecimal digit.

Other Languages

Java does not support eight-digit universal character names.

Coding Guidelines

external identifier 283
significant characters

This form of UCN counts toward a greater number of significant characters in identifiers with external linkage and therefore is not the preferred representation. However, the developer may not have any control over the method used by an editor to represent UCNs. Given that characters from the majority of human languages can be represented using four-digit short identifiers, eight-digit short identifiers are not likely to be needed. If the development environment offers a choice of representations, use of four-digit short identifiers is likely to result in more significant characters being retained in identifiers having external linkage.

Similarly, the universal character name \u*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000*nnnn*). 819

Commentary

It was possible to represent all of the characters specified by versions 1 and 2 of the Unicode-sponsored character set using four-digit short identifiers. Version 3 introduced characters whose representation value requires more than four digits.

Other Languages

Java only supports this form of four-digit universal character names.

footnote 62

62) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF-16). 820

Commentary

basic character set 215

Requiring that characters in the basic character set not be represented using UCN notation helps guarantee that existing tools (e.g., editors) continue to be able to process source files.

The control characters may have special meaning for some tools that process source files (e.g., a communications program used for sending source down a serial link).

C++

The C++ Standard does not make this observation.

footnote 63

63) Short identifiers for characters were first specified in ISO/IEC 10646–1/AMD9:1997. 821

Commentary

This amendment appeared eight years after the first publication of the C Standard (which was made by ANSI in 1989).

6.4.4 Constants

822 constant syntax

```
constant:
        integer-constant

        floating-constant
        enumeration-constant
        character-constant
```

Commentary

A *constant* differs from a *constant-expression* in that it consists of a single token. The term *literal* is often used by developers to refer to what the C Standard calls a *constant* (technically the only literals C contains are string literals). There is a more general usage of the term *constant* to mean something whose value does not change. What the C Standard calls a *constant-expression* developers often shorten to *constant*.

1322 constant expression syntax

895 string literal syntax

C++

> *21) The term "literal" generally designates, in this International Standard, those tokens that are called "constants" in ISO C.*

Footnote 21

The C++ Standard also includes *string-literal* and *boolean-literal* in the list of literals, but it does not include enumeration constants in the list of literals. However:

> *The identifiers in an* `enumerator-list` *are declared as constants, and can appear wherever constants are required.*

7.2p1

The C++ terminology more closely follows common developer terminology by using *literal* (a single token) and *constant* (a sequence of operators and literals whose value can be evaluated at translation time). The value of a literal is explicit in the sequence of characters making up its token. A constant may be made up of more than one token or be an identifier. The operands in a constant have to be evaluated by the translator to obtain its result value. C uses the more easily confused terminology of *integer-constant* (a single token) and *constant-expression* (a sequence of operators, *integer-constant* and *floating-constant* whose value can be evaluated at translation time).

Other Languages

Languages that support types not supported by C (e.g., instance sets) sometimes allow constants having these types to be specified (e.g., in Pascal `['a', 'd']` represents a set containing two characters). Fortran supports complex literal constants (e.g., `(1.0, 2.0)` represents the complex number $1.0 + 2.0i$)

Many languages do not support (e.g., Java until version 1.5) some form of *enumeration-constant*.

Coding Guidelines

Constants are the mechanism by which numeric values are written into source code. The term *constant* is used because the numeric values do not change during program execution (and are known at translation time;

although in some cases a person reading the source may only know that the value used will be one of a list of possible values because the definition of a macro may be conditional on the setting of some translation time option— for instance, `-D`).

macro 1931 object-like

The use of constants in source code creates a number of possible maintenance issues, including:

- A constant value, representing some quantity, often needs to occur in multiple locations within source code. Searching for and replacing all occurrences of a particular numeric value in the code is an error prone process. It is not possible, for instance, to know that all `15`s occurring in the source code have the same semantic association and some may need to remain unchanged. (Your author was once told by a developer, whose source contained lots of `15`s, that the UK government would never change value-added tax from 15%; it is now 17.5%.)
- On encountering a constant in the source, a reader usually needs to deduce its semantic association (either in the application domain or its internal algorithmic function). While its semantics may be very familiar to the author of the source, the association between value and semantics may not be so readily made by later readers.
- A cognitive switch may need to be made because of the representation used for the constant (e.g., floating point, hexadecimal integer, or character constant).

cognitive 0 switch

symbolic name

One solution to these problems is to use an identifier to give a symbolic name[822.1] to the constant, and to use that symbolic name wherever the constant would have appeared in the source. Changes to the value of the constant can then be made by a single modification to the definition of the identifier and a well-chosen name can help readers make the appropriate semantic association. The creation of a symbolic name provides two pieces of information:

1. *The property represented by that symbolic name.* For instance, the maximum value of a particular type (`INT_MAX`), whether an implementation supports some feature (`__STDC_IEC_559__`), a means of specifying some operation (`SEEK_SET`), or a way to obtain information (`FE_OVERFLOW`).
2. *A method of operating on the symbolic name to access the property it represents.* For instance, arithmetic operations (`INT_MAX`), testing in a conditional preprocessing directive (`__STDC_IEC_559__`), passing as an argument to a library function (`SEEK_SET`); passing as an argument to a library function, possibly in combination with other symbolic names (`FE_OVERFLOW`).

INT_MAX 318 2015
__STDC_IEC_559__ macro

Operating on symbolic names involves making use of representation information. (Assignment, or argument passing, is the only time that representation might not be an issue.) The extent to which the use of representation information will be considered acceptable will depend on the symbolic name. For instance, `FE_OVERFLOW` appearing as the operand of a bitwise operator is to be expected, but its appearance as the operand of an arithmetic operator would be suspicious.

The use of symbolic names is rarely seen by developers, as applying to all constants that occur in source code. In some cases the following are claimed:

- The constants are so sufficiently well-known that there is no need to give them a name.
- The number of occurrences of particular constants is not sufficient to warrant creating a name for them.
- Operations involving some constant values occur so frequently that their semantic associations are obvious to developers; for instance, assigning `0` or adding `1`.

It is true that not all numeric values are meaningless to everybody. A few values are likely to be universally known (at least to Earth-based developers). For instance, there are 60 seconds in a minute, 60 minutes in an

[822.1] In some cases the linguistically more correct terminology would be *iconic name*.

hour, and 24 hours in a day. The value 24 occurring in an expression involving time is likely to represent hours in a day. Many values will only be well known to developers working within a given application domain, such as atomic physics (e.g., the value 6.6261E-34). Between these extremes are other values; for instance, 3.14159 will be instantly recognized by developers with a mathematics background. However, developers without this background may need to think about what it represents. There is the possibility that developers who have grown up surrounded by other mathematically oriented people will be completely unaware that others do not recognize the obvious semantic association for this value.

A constant having a particular semantic association may only occur once in the source. However, the issue is not how many times a constant having a particular semantic association occurs, but how many times the particular constant value occurs. The same constant value can appear because of different semantic associations. A search for a sequence of digits (a constant value) will locate all occurrences, irrespective of semantic association.

While an argument can always be made for certain values being so sufficiently well-known that there is no benefit in replacing them by identifiers, the effort/time taken in discussions on what values are sufficiently well-known to warrant standing on their own, instead of an identifier, is likely to be significantly greater than the sum total of all the extra one seconds, or so, taken to type the identifier.

The constant values 0 and 1 occur very frequently in source code (see Figure 825.1). Experience suggests that the semantic associations tend to be that of assigning an initial value in the case of 0 and accessing a preceding or following item in the case of 1. The coding guideline issues are discussed in the subsections that deal with the different kinds of constants (e.g., integer, or floating).

What form of definition should a symbolic name denoting constant value have? Possibilities include the following:

- *Macro names.* These are seen by developers as being technically the same as constants in that they are replaced by the numeric value of the constant during translation (there can also be an unvoiced bias toward perceived efficiency here).

- *Enumeration constants.* The purpose of an enumerated type is to associate a list of constants with each other. This is not to say the definition of an enumerated type containing a single enumeration constant should not occur, but this usage would be unusual. Enumeration constants share the same unvoiced developer bias as macro names— perceived efficiency.

517 enumeration
set of named constants

- *Objects initialized with the constant.* This approach is advocated by some coding guideline documents for C++. The extent to which this is because an object declared with the **const** qualifier really is constant and a translator need not allocate storage for it, or because use of the preprocessor (often called the C preprocessor, as if it were not also in C++) is frowned on in the C++ community and is left to the reader to decide.

The enumeration constant versus macro name issue is discussed in detail elsewhere.

517 enumeration
set of named constants

What name to choose? The constant 6.6261E-34 illustrates another pitfall. Planck's constant is almost universally represented, within the physics community, using the letter *h* (sometimes with embellishments, e.g., $\hbar$). A developer might be tempted to make use of this idiom to name the value, perhaps even trying to find a way of using UCNs to obtain the appropriate h. The single letter h probably gives no more information than the value. The name PLANCK_CONSTANT is self-evident. The developer attitude— anybody who does not know what 6.6261E-34 represents has no business reading the source— is not very productive or helpful.

Table 822.1: Occurrence of different kinds of constants (as a percentage of all tokens). Based on the visible form of the `.c` and `.h` files.

Kind of Constant	`.c` files	`.h` files
character-constant	0.16	0.06
integer-constant	6.70	20.79
floating-constant	0.02	0.20
string-literal	1.02	0.74

Constraints

constant representable in its type

The value of a constant shall be in the range of representable values for its type. 823

Commentary

constant 824 type determined by form and value

This is something of a circular definition in that a constant's value is also used to determine its type. The lexical form of a constant is also a factor in determining which of a number of possible types it may take. An unsuffixed constant that is too large to be represented in the type **`long long`**, or a suffixed constant that is larger than the type with the greatest rank applicable to that suffix, violates this requirement (unless there is some extended integer type supported by the implementation into whose range the value falls).

It can be argued that all floating constants are in range, if the implementation supports $\pm\infty$.

enumeration constant 1440 representable in int

There is a similar constraint for enumeration constants

C++

The C++ Standard has equivalent wording covering *integer-literals* (2.13.1p3) *character-literals* (2.13.2p3) and *floating-literals* (2.13.3p1). For *enumeration-literals* their type depends on the context in which the question is asked:

7.2p4 *Following the closing brace of an* `enum-specifier`*, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value.*

7.2p5 *The underlying type of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.*

Other Languages

Most languages have a similar requirement, even those supporting a single integer or floating type.

Common Implementations

Some implementations use the string-to-integer conversions provided by the library, while others prefer the flexibility (and fuller control of error recovery) afforded by specially written code. Parker[1055] describes the minimal functionality required.

Example

```
1  char ch = '\0\0\0\0y';
2
3  float f_1 = 1e99999999999999999999999999999999999999999999999;
4  float f_2 = 0e99999999999999999999999999999999999999999999999;
5  float f_3 = 1e-99999999999999999999999999999999999999999999999; /* Approximately zero. */
6  float f_4 = 0e-99999999999999999999999999999999999999999999999; /* Exact zero. */
7
8  short s_1 = 99999999999999999999999999999999999999999999999;
9  short s_2 = 99999999999999999999 / 99999999999999999999;
```

The integer constant 10000000000000000000L would violate this constraint on an implementation that represented the type **long long** in 64 bits. The use of an L suffix precludes the constant being given the type **unsigned long long**.

Semantics

824 Each constant ~~has a type, determined by its form and value, as detailed later.~~ shall have a type and the value of a constant shall be in the range of representable values for its type.

constant type determined by form and value

Commentary

Just as there are different floating and integer object types, the possible types that constants may have is not limited to a single type.

836 integer constant possible types

It is a constraint violation for a constant to occur during translation phrase 7 without a type.

136 translation phase 7

The requirement that a constant be in the range of representable values for its type is a requirement on the implementation.

841 integer constant no type

The wording was changed by the response to DR #298.

C++

The type of an integer literal depends on its form, value, and suffix. 2.13.1p2

The type of a floating literal is ***double*** *unless explicitly specified by a suffix. The suffixes* `f` *and* `F` *specify* ***float****, the suffixes* `l` *and* `L` *specify* ***long double****.* 2.13.3p1

There are no similar statements for the other kinds of literals, although C++ does support suffixes on the floating types. However, the syntactic form of string literals, character literals, and boolean literals determines their type.

Coding Guidelines

The type of a constant, unlike object types, can vary between implementations. For instance, the integer constant 40000 can have either the type **int** or **long int**. The suffix on the integer constant 40000u only ensures that it has one of the listed unsigned integer types. The coding guideline issues associated with the possibility that the type of a constant can vary between implementations is discussed elsewhere.

835 integer constant type first in list

6.4.4.1 Integer constants

integer constant syntax

825

```
integer-constant:
                decimal-constant integer-suffix_opt
                octal-constant integer-suffix_opt
                hexadecimal-constant integer-suffix_opt
decimal-constant:
                nonzero-digit
                decimal-constant digit
octal-constant:
                0
                octal-constant octal-digit
hexadecimal-constant:
                hexadecimal-prefix hexadecimal-digit
                hexadecimal-constant hexadecimal-digit
hexadecimal-prefix: one of
```

```
                0x  0X
nonzero-digit: one of
                1  2  3  4  5  6  7  8  9

octal-digit: one of
                0  1  2  3  4  5  6  7
hexadecimal-digit: one of
                0  1  2  3  4  5  6  7  8  9
                a  b  c  d  e  f
                A  B  C  D  E  F
integer-suffix:
                unsigned-suffix long-suffix_opt
                unsigned-suffix long-long-suffix
                long-suffix unsigned-suffix_opt
                long-long-suffix unsigned-suffix_opt
unsigned-suffix: one of
                u  U
long-suffix: one of
                l  L
long-long-suffix: one of
                ll  LL
```

Commentary

translation phase 7 (136)

constant expression syntax (1322)

Integer constants are created in translation phase 7 when the preprocessing tokens *pp-number* are converted into tokens denoting various forms of *constant*. *Integer-constant*s always denote positive values. The character sequence `-1` consists of the two tokens `{-}` `{1}`, a constant expression.

An *integer-suffix* can be used to restrict the set of possible types the constant can have, it also specifies the lowest rank an integer constant may have (which for *ll* or *LL* leaves few further possibilities). The *U*, or *u*, suffix indicates that the integer constant is unsigned.

All translation time integer constants are nonnegative. The character sequence `-1` consists of the token sequence unary minus followed by the *decimal-constant* `1`. Support for translation time negative constants in the lexical grammar would create unjustified complexity by requiring lexers to disambiguate binary from unary operators uses in, for instance: `X-1`.

C90

Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is new in C99.

C++

The C++ syntax is identical to the C90 syntax.

Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is not available in C++.

Common Implementations

Some implementations specify that the suffix *0b* (or *0B*) denotes an integer constant expressed in binary notation. Over the years the C Committee received a number of requests for such a suffix to be added to the C Standard. The Committee did not see sufficient utility for this suffix to be included in C99. The C embedded systems TR specifies *h* and *H* to denote the types **short frac** or **short accum**, and one of *k*, *K*, *r*, and *R* to denote a fixed-point type.

Embedded C TR (18)

The IBM ILE C compiler[617] supports a packed decimal data type. The suffix *d* or *D* may be used to specify that a literal has this type. Microsoft C supports the suffixes *i8*, *i16*, *i32*, and *i64* denoting integer

constants having the types **byte** (an extension), **short**, **int**, and **__int64**, respectively.

Other Languages

Although Ada supports integer constants having bases between 1 and 36 (e.g., 2#1101 is the binary representation for 10#13), few other languages support the use of suffixes. Ada also supports the use of underscores within an *integer-constant* to make the value more readable.

Coding Guidelines

A study by Brysbaert[174] found that the time taken for a person to process an Arabic integer between 1 and 99 was a function of the logarithm of its magnitude, the frequency of the number (based on various estimates of its frequency of occurrence in everyday life; see Dorogovtsev et al[367] for measurements of numbers appearing in web pages), and sometimes the number of syllables in the spoken form of the value. Subject response times varied from approximately 300 ms for values close to zero, to approximately 550 ms for values in the nineties.

Experience shows that the *long-suffix* `l` is often visually confused with the *nonzero-digit* `1`.[825.1]

Cg 825.1 If a *long-suffix* is required, only the form `L` shall be used.

Cg 825.2 If a *long-long-suffix* is required, only the form `LL` shall be used.

As previously pointed out, constants appearing in the visible form of the source often signify some quantity with real world semantics attached to it. However, uses of the integer constants 0 and 1 in the visible source often have no special semantics associated with their usage. They also represent a significant percentage of the total number of integer constants in the source code (see Figure 825.1). The frequency of occurrence of these values (most RISC processors dedicate a single register to permanently hold the value zero) comes about through commonly seen program operations. These operations include: code to count the number of occurrences of entities, or that contain loops, or index the previous or next element of an array (not that 0 or 1 could not also have similar semantic meaning to other constant values).

822 constant syntax

A blanket requirement that all integer constants be represented in the visible source by symbolic names fails to take into account that a large percentage of the integer constants used in programs have no special meaning associated with them. In particular the integer constants 0 and 1 occur so often (see Figure 825.1) that having to justify why each of them need not be replaced by a symbolic name would have a high cost for an occasional benefit.

Rev 825.3 No integer constant, other than 0 and 1, shall appear in the visible source code, other than as the sole preprocessing token in the body of a macro definition or in an enumeration definition.

Some developers are sloppy in the use of integer constants, using them where a floating constant was the appropriate type. The presence of a period makes it explicitly visible that a floating type is being used. The general issue of integer constant conversions is discussed elsewhere.

835.2 integer constant with suffix, not immediately converted

Example

The character sequence `123xyz` is tokenized as {`123xyz`}, a *pp-number*. This is not a valid integer constant.

927 pp-number syntax

Usage

Having some forms of constant tokens (also see Figure 842.1) follow Benford's law[574] would not be surprising because the significant digits of a set of values created by randomly sampling from a variety of

integer constant usage

[825.1] While the visual similarity between alphabetic letters has been experimentally measured your author is not aware of any experiment that has measured the visually similarity of digits with letters.

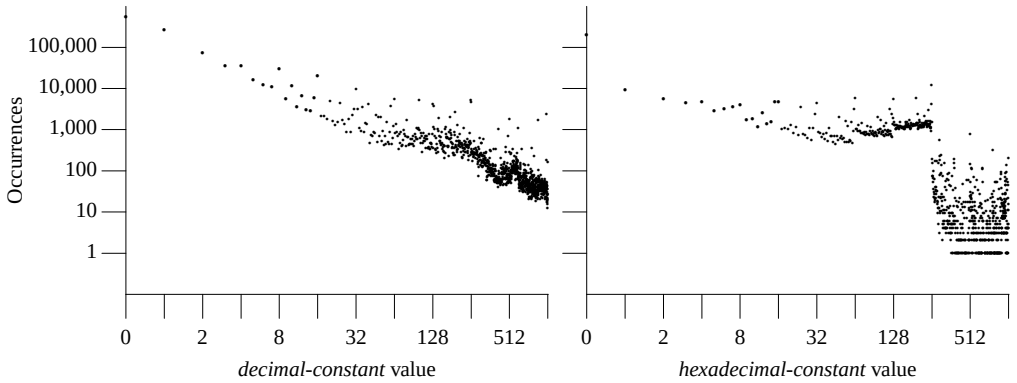


Figure 825.1: Number of integer constants having the lexical form of a *decimal-constant* (the literal 0 is also included in this set) and *hexadecimal-constant* that have a given value. Based on the visible form of the .c and .h files.

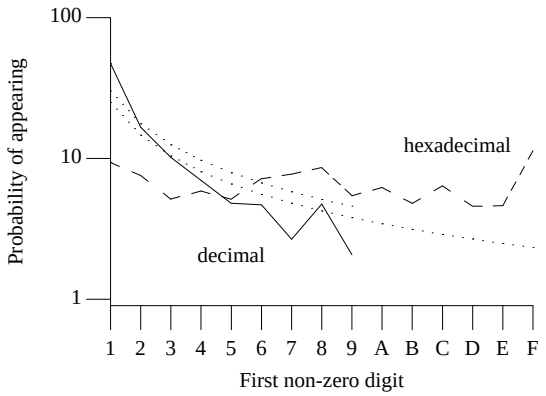


Figure 825.2: Probability of a *decimal-constant* or *hexadecimal-constant* starting with a particular digit; based on .c files. Dotted lines are the probabilities predicted by Benford's law (for values expressed in base 10 and base 16), i.e., $\log(1 + d^{-1})$, where d is the numeric value of the digit.

different distributions converges to a logarithmic distribution (i.e., Benford's law).[573] While the results for *decimal-constant* (see Figure 825.2) may appear to be a reasonable fit, applying a chi-squared test shows the fit to be remarkably poor (χ^2 = 132,398). The first nonzero digit of *hexadecimal-constant*s appears to be approximately evenly distributed.

Table 825.1: Occurrence of various kinds of *integer-constant*s (as a percentage of all integer constants; note that zero is included in the *decimal-constant* count rather than the *octal-constant* count). Based on the visible form of the .c and .h files.

Kind of *integer-constant*	.c files	.h files
decimal-constant	64.1	17.8
hexadecimal-constant	35.8	82.1
octal-constant	0.1	0.2

Table 825.2: Occurrence of various *integer-suffix* sequences (as a percentage of all *integer-constants*). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	`.c` files	`h.` files	Suffix Character Sequence	`.c` files	`.h` files
none	99.6850	99.5997	Lu/lU	0.0005	0.0001
U/u	0.0298	0.0198	LL/lL/ll	0.0072	0.0022
L/l	0.1378	0.2096	ULL/uLl/ulL/Ull	0.0128	0.0061
U/uL/ul	0.1269	0.1625	LLU/lLu/LlU/llu	0.0000	0.0000

Table 825.3: Common token pairs involving *integer-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, *integer-constant*	42.9	56.5	**(** *integer-constant*	2.8	3.4
integer-constant **]**	6.4	44.4	**==** *integer-constant*	25.5	2.0
integer-constant ,	58.2	44.2	**return** *integer-constant*	18.6	1.9
integer-constant **;**	14.1	12.1	**+** *integer-constant*	33.7	1.9
integer-constant **)**	14.2	11.7	**&** *integer-constant*	30.6	1.5
integer-constant **#**	1.4	9.1	identifier *integer-constant*	0.3	1.5
= *integer-constant*	19.6	9.0	**-** *integer-constant*	44.0	1.3
[*integer-constant*	39.3	5.6	**<** *integer-constant*	40.0	1.3
integer-constant **}**	1.2	4.4	**{** *integer-constant*	4.2	1.2
-v *integer-constant*	69.0	4.1			

A study by Pollmann and Jansen[1101] analyzed occurrences of related pairs of numerals (e.g., "two or three books") in written (Dutch) text. They found that pairs of numerals often followed what they called *ordering rules*, which were (for the number pair x and y):

- x has to be smaller than y
- x or y has to be *round* (i.e., *round* numbers include the numbers 1 to 20 and the multiples of five)
- the difference between x and y has to be a favorite number. (These include: $10^n \times (1, 2, ½, \text{or } ¼)$ for any value of n.)

Description

826 An integer constant begins with a digit, but has no period or exponent part. integer constant

Commentary

A restatement of information given in the Syntax clause.

827 It may have a prefix that specifies its base and a suffix that specifies its type.

Commentary

A suffix need not uniquely determine an integer constants type, only the lowest rank it may have. There is no suffix for specifying the type **int**, or any integer type with rank less than **int** (although implementations may provide these as an extension).

The base document did not specify any suffixes; they were introduced in C90. 1 base document

Other Languages

A few other languages also support some kind of suffix, including C++, Fortran, and Java.

Coding Guidelines

terminology integer constant

Developers do not normally think in terms of an integer constant having a prefix. The term *integer constant* is often used to denote what the standard calls a *decimal constant*, which corresponds to the common case. When they occur in source, both octal and hexadecimal constants are usually referred to by these names, respectively. The benefits of educating developers to use the terminology *decimal constant* instead of *integer constant* are very unlikely to exceed the cost.

---

decimal constant

A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. 828

Commentary

A restatement of information given in the Syntax clause.

Coding Guidelines

The constant 0 is, technically, an octal constant. Some guideline documents use the term decimal constant in their wording, overlooking the fact that, technically, this excludes the value 0. The guidelines given in this book do not fall into this trap, but anybody who creates a modified version of them needs to watch out for it.

---

octal constant

An octal constant consists of the prefix **0** optionally followed by a sequence of the digits **0** through **7** only. 829

Commentary

A restatement of information given in the Syntax clause. An octal constant is a natural representation to use when the value held in a single byte needs to be displayed (or read in) and the number of output indicators (or input keys) is limited (only eight possibilities are needed). For instance, a freestanding environment where the output device can only represent digits. The users of such input/output devices tend to be technically literate.

Other Languages

A few other languages (e.g., Java and Ada) support octal constants. Most do not.

Common Implementations

K&R C supported the use of the digits 8 and 9 in octal constants (support for this functionality was removed during the early evolution of C[1180] although some implementations continue to support it[600, 1075]). They represented the values 10 and 11, respectively.

Coding Guidelines

Octal constants are rarely used (approximately 0.1% of all *integer-constant*s, not counting the value 0). There seem to be a number of reasons why developers occasionally use octal constants:

- A long-standing practice that arguments to calls to some Unix library functions use octal constants to indicate various attributes (e.g., `open(file, O_WRONLY, 0666)`). The introduction, by POSIX in 1990, of identifiers representing these properties has not affected many developers' coding habits. The value 0666, in this usage, could be said to be treated like a symbolic identifier.
- Cases where it is sometimes necessary to think of a bit pattern in terms of its numeric value. Bit patterns are invariably grouped into bytes, making hexadecimal an easier representation to manipulate (because its visual representation is easily divisible into bytes and half bytes). However, mental arithmetic involving octal digits is easier to perform than that with hexadecimal digits. (There are fewer items of information that need to be remembered and people have generally automated the processing of digits, but conscious effort is needed to map the alphabetic letters to their numeric equivalents.)
- The values are copied from an external source; for instance, tables of measurements printed in octal.

escape sequence 873 octal digits

There are no obvious reasons for recommending the use of octal constants over decimal or hexadecimal constants (there is a potential advantage to be had from using octal constants).

830 A hexadecimal constant consists of the prefix **0x** or **0X** followed by a sequence of the decimal digits and the letters **a** (or **A**) through **f** (or **F**) with values 10 through 15 respectively.

hexadecimal constant

Commentary

A restatement of information given in the Syntax clause. A hexadecimal constant provides a natural way of denoting a value, occupying one or more 8-bit bytes, when its underlying representation is of interest. Each digit in a hexadecimal constant represents four binary digits, a nibble.

Other Languages

Many languages support the representation of hexadecimal constants in source code. The prefix character *$* (not available in the C basic source character set) is used almost as often, if not more so, than the *0x* form of prefix.

Coding Guidelines

In many cases a hexadecimal constant is not thought about, by developers, in terms of being a number but as representing a pattern of bits (perhaps even having an internal structure to it). For instance, in a number of applications objects and constants have values that are more meaningfully thought about in terms of powers of two rather than powers to ten. In such cases a constant appearing in the source as a hexadecimal constant is more easily appreciated (in terms of the sums of the powers of two involved and by which powers of two it differs from other constants) than if expressed as a decimal constant.

825 integer constant usage

Measurements of constant use in source code show that usage patterns for hexadecimal constants are different from decimal constants. The probability of a particular digit being the first nonzero digit in a hexadecimal constant is roughly constant, while the probability distribution of this digit in a decimal constant decreases with increasing value (a ch-squared analysis gives a very low probability of it matching Benford's law). Also the sequence of value digits in a *hexadecimal-constant* (see Table 830.1) almost always exactly corresponds to the number of nibbles in either a character type, **short**, **int**, or **long**.

0 automatization

A study by Logan and Klapp[863] used *alphabet arithmetic* (e.g., $A + 2 = C$) to investigate how extended practice and rote memorization affected automaticity. For inexperienced subjects who had not memorized any addition table, the results showed that the time taken to perform the addition increased linearly with the value of the digit being added. This is consistent with subjects counting through the letters of the alphabet to obtain the answer. With sufficient practice subjects performance not only improved but became digit-independent. This is consistent with subjects recalling the answer from memory; the task had become automatic.

The *practice* group of subjects were given a sum and had to produce the answer. The *memorization* group of subjects were asked to memorise a table of sums and there answers, (e.g., $A + 2 = C$). In both cases the results showed that performance was proportional to the number of times each question/answer pair had been encountered, not the total amount of time spent.

Arithmetic involving hexadecimal constants differs from that involving decimal constants in that developers will have had much less experience in performing it. The results of the Logan and Klapp study show that the only way for developers to achieve the same level of proficiency is to commit the hexadecimal addition table to memory. Whether the cost of this time investment has a worthwhile benefit is unknown.

Table 830.1: Occurrence of *hexadecimal-constant*s containing a given number of digits (as a percentage of all such constants). Based on the visible form of the `.c` files.

Digits	Occurrence	Digits	Occurrence	Digits	Occurrence	Digits	Occurrence
0	0.003	5	0.467	10	0.005	15	0.000
1	1.092	6	0.226	11	0.001	16	0.209
2	59.406	7	0.061	12	0.001		
3	1.157	8	2.912	13	0.000		
4	34.449	9	0.010	14	0.000		

Semantics

The value of a decimal constant is computed base 10; 831

Commentary

C supports the representation of constants in the base chosen by evolution on planet Earth.

that of an octal constant, base 8; 832

Commentary

unsigned 593
integer types
object representation

The C language requires the use of binary representation for the integer types. The use of both base 8 and base 16 visual representations of binary information has been found to be generally more efficient, for people, than using a binary representation. Developers continue to debate the merits of one base over another. Both experience with using one particular base and the kind of application domain affect preferences.

that of a hexadecimal constant, base 16. 833

Commentary

The correct Latin prefix is *sex*, giving *sexadecimal*. It has been claimed that this term was considered too racey by IBM who adopted *hexadecimal* (*hex* is the equivalent Greek prefix, the Latin *decimal* being retained) in the 1960s to replace it (the term was used in 1952 by Carl-Eric Froeberg in a set of conversion tables).

The lexically first digit is the most significant. 834

Commentary

The Arabic digits in a constant could be read in any order. In Arabic, words and digits are read/written right-to-left (least significant to most significant in the case of numbers). The order in which Arabic numerals are written was exactly copied by medieval scholars, except that they interpreted them using the left-to-right order used in European languages.

integer constant type first in list

The type of an integer constant is the first of the corresponding list in which its value can be represented. 835

Commentary

translation phase 136
7

This list only applies to those *pp-numbers* that are converted to *integer-constant* tokens as part of translation phase 7. Integer constants in **#if** preprocessor directives always have type `intmax_t`, or `uintmax_t` (in C90 they had type **long** or **unsigned long**).

Other Languages

In Java integer constants have type **int** unless they are suffixed with *l*, or *L*, in which case they have type **long**. Many languages have a single integer type, which is also the type of all integer constants.

Coding Guidelines

The type of an integer constant may depend on the characteristics of the host on which the program executes and the form used to express its value. For instance, the integer constant 40000 may have type **int** or **long int** (depending on whether **int** is represented in more than 16 bits, or in just 16 bits). The hexadecimal constant 0x9C40 (40000 decimal) may have type **int** or **unsigned int** (depending on the whether **int** is represented in more than 16 bits, or in just 16 bits).

object 480.1
int type only

For objects having an integer type there is a guideline recommending that a single integer type always be used (the type **int**). However, integer constants never have a type whose rank is less than **int** and so the developer issues associated with the integer promotions do not apply. It makes no sense for a coding guideline to recommend against the use of an *integer-constant* whose value is not representable in the type **int** (a developer is unlikely to use such a value without the application requiring it).

The possibility that the type of an integer constant can vary between implementations and platforms creates a portability cost. There is also the potential for incorrect developer assumptions about the type of an

integer constant, leading to additional maintenance costs. The specification of a guideline recommendation is complicated by the fact that C does not support a suffix that specifies the type **int** (or its corresponding unsigned version). This means it is not possible to specify that a constant, such as 40000, has type **int** and expect a diagnostic to appear when using a translator that gives it the type **long**.

Cg 835.1 An unsuffixed *integer-constant* having a value greater than 32767 shall be treated, for the purposes of these guideline recommendations, as if its lexical included a suffix specifying the type **int**.

An integer constant containing a suffix is generally taken as a statement of intent by the developer. A suffixed integer constant that is immediately converted to another type is suspicious.

Cg 835.2 An integer constant containing a suffix shall not be immediately converted to another type.

Dev 835.2 The use of a macro defined in a system header may be immediately cast to another type.

Dev 835.2 The use of a macro defined in a developer written system header may be immediately cast to another type, independent of how the macro is implemented.

Dev 835.2 The body of a macro may convert, to an integer type, one of the parameters of that macro definition.

Dev 835.2 If the range of values supported by the type **unsigned short**, or **unsigned char**, is the same as that supported by **unsigned int**, an integer constant containing an unsigned suffix may be converted to those types.

Is there anything to be gained from recommending that integer constants less than 32767 be suffixed rather than implicitly converted to another type? The original type of such an integer constant is obvious to the reader and a conversion to a type for which the standard provides a suffix will not change its value; the real issue is developer expectation. Expectation can become involved through the semantics of what the constant represents. For instance, a program that manipulates values associated with the ISO 10646 Standard may store these values in objects that always have type **unsigned int**. This usage can lead to developers learning (implicitly or explicitly) that objects manipulating these semantic quantities have type **unsigned int**, creating an expectation that all such quantities have this type. Expectations on the sign of an operand can show up as a difference between actual and expected behavior; for instance, the following expression checks if any bits outside of the least significant octet are set: `~F00_char > 0x00ff`. It only works if the left operand has an unsigned type. (If it has a signed type, setting the most significant bit will cause the result to be negative.) If the identifier `F00_char` is a macro whose body is a constant integer having a signed type, developer expectations will not have been met.

0 implicit learning

In those cases where developers have expectations of an operand having a particular type, use of a suffix can help ensure that this expectation is met. If the integer constant appears in the visible source at the point its value is used, developers can immediately deduce its type. An integer constant in the body of a macro definition or as an argument in a macro invocation are the two circumstances where type information is not immediately apparent to readers of the source. (The integer constant is likely to be widely separated from its point of use in an expression.)

The disadvantage of specifying a suffix on an integer constant because of the context in which it is used is that the applicable type may change. The issues involved with implicit conversion versus explicit conversion are discussed elsewhere. An explicit cast, using a typedef name rather than a suffix, is more flexible in this regard.

654 implicit conversion

Use of a suffix not defined by the standard, but provided by the implementation, is making use of an extension. Does this usage fall within the guideline recommendation dealing with use of extensions, or is it sufficiently useful that a deviation should be made for it? Suffixes are a means for the developer to specify type information on integer constants. Any construct that enables the developer to provide more information

95.1 extensions cost/benefit

is usually to be encouraged. While there are advantages to this usage, at the time of this writing insufficient experience is available on the use of suffixes to know whether the advantages outweigh the disadvantages. A deviation against the guideline recommendation might be applicable in some cases.

Dev 95.1 Any integer constant suffix supported by an implementation may be used.

Table 835.1: Occurrence of *integer-constant*s having a particular type (as a percentage of all such constants; with the type denoted by any suffix taken into account) when using two possible representations of the type **int** (i.e., 16- and 32-bit). Based on the visible form of the `.c` and `.h` files.

Type	16-bit **int**	32-bit **int**
int	94.117	99.271
unsigned int	3.493	0.414
long	1.805	0.118
unsigned long	0.557	0.138
other-types	0.029	0.059

integer constant possible types

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	**int** **long int** **long long int**	**int** **unsigned int** **long int** **unsigned long int** **long long int** **unsigned long long int**
u or **U**	**unsigned int** **unsigned long int** **unsigned long long int**	**unsigned int** **unsigned long int** **unsigned long long int**
l or **L**	**long int** **long long int**	**long int** **unsigned long int** **long long int** **unsigned long long int**
Both **u** or **U** and **l** or **L**	**unsigned long int** **unsigned long long int**	**unsigned long int** **unsigned long long int**
ll or **LL**	**long long int**	**long long int** **unsigned long long int**
Both **u** or **U** and **ll** or **LL**	**unsigned long long int**	**unsigned long long int**

Commentary

The lowest rank that an integer constant can have is type **int**. This list contains the standard integer types only, giving preference to these types. Any supported extended integer type is considered if an appropriate type is not found from this list.

C90

The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal: ***int****,* ***long int****,* ***unsigned long int****; unsuffixed octal or hexadecimal:* ***int****,* ***unsigned int****,* ***long int****,* ***unsigned long int****; suffixed by the letter* `u` *or* `U`*:* ***unsigned int****,* ***unsigned long int****; suffixed by the letter* `l` *or* `L`*:* ***long int****,* ***unsigned long int****; suffixed by both the letters* `u` *or* `U` *and* `l` *or* `L`*:* ***unsigned long int****.*

Support for the type **long long** is new in C99.

The C90 Standard will give a sufficiently large decimal constant, which does not contain a *u* or *U* suffix— the type **unsigned long**. The C99 Standard will never give a decimal constant that does not contain either of these suffixes— an unsigned type.

Because of the behavior of C++, the sequencing of some types on this list has changed from C90. The following shows the entries for the C90 Standard that have changed.

Suffix	Decimal Constant
none	**int** **long int** **unsigned long int**
l or **L**	**long int** **unsigned long int**

Under C99, the none suffix, and *l* or *L* suffix, case no longer contain an unsigned type on their list.

A decimal constant, unless given a *u* or *U* suffix, is always treated as a signed type.

C++

2.13.1p2

If it is decimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***long int****; if the value cannot be represented as a* ***long int****, the behavior is undefined. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented:* ***int***, ***unsigned int***, ***long int***, ***unsigned long int****. If it is suffixed by* u *or* U*, its type is the first of these types in which its value can be represented:* ***unsigned int***, ***unsigned long int****. If it is suffixed by* l *or* L*, its type is the first of these types in which its value can be represented:* ***long int***, ***unsigned long int****. If it is suffixed by* ul, lu, uL, Lu, Ul, lU, UL, *or* LU*, its type is* ***unsigned long int****.*

The C++ Standard follows the C99 convention of maintaining a decimal constant as a signed and never an unsigned type.

The type **long long**, and its unsigned partner, is not available in C++.

There is a difference between C90 and C++ in that the C90 Standard can give a sufficiently large decimal literal that does not contain a *u* or *U* suffix— the type **unsigned long**. Neither the C++ or C99 Standard will give a decimal constant that does not contain either of these suffixes— an unsigned type.

Other Languages

In Java hexadecimal and octal literals always have a signed type and denote a negative value if the high-order bit, for their type, is set. The literal 0xcafebabe has decimal value -889275714 and type **int** in Java, and decimal value 3405691582 and type **unsigned int** or **unsigned long** in C.

---

837 If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value.

Commentary

For an implementation to support an integer constant which is not representable by any standard integer type, requires that it support an extended integer type that can represent a greater range of values than the types **long long** or **unsigned long long**.

C90

Explicit support for extended types is new in C99.

C++

The C++ Standard allows new object types to be created. It does not specify any mechanism for giving literals these types.

A C translation unit that contains an integer constant that has an extended integer type may not be accepted by a conforming C++ translator. But then it may not be accepted by another conforming C translator either. Support for the construct is implementation-defined.

Other Languages

Very few languages explicitly specify potential implementation support for extended integer types.

Common Implementations

integer constant syntax 825

In some implementations it is possible for an integer constant to have a type with lower rank than those given on this list.

Coding Guidelines

extensions cost/benefit 95.1

Source containing an integer constant, the value of which is not representable in one of the standard integer types, is making use of an extension. The guideline recommendation dealing with use of extensions is applicable here. If it is necessary for a program to use an integer constant having an extended integer type, the deviation for this guideline specifies how this usage should be handled. The issue of an integer constant being within the range supported by a standard integer type on one implementation and not within range on another implementation is discussed elsewhere.

integer constant greater than 32767 835.1

If all of the types in the list for the constant are signed, the extended integer type shall be signed. 838

Commentary

This is a requirement on the implementation. This requirement applies to the standard integer types. By requiring that any extended integer type follow the same rule, the standard is preserving the idea that decimal constants are signed unless they contain an unsigned suffix. All of the types in the list are signed if the lexical representation is a decimal constant without a suffix, or a decimal constant whose suffix is not *u* or *U*.

If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. 839

Commentary

This is a requirement on the implementation. The types in the list are all unsigned if the integer constant contains a *u* or *U* suffix.

If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. 840

Commentary

Both signed and unsigned types only occur if octal or hexadecimal notation is used, and no *u* or *U* suffix appears in the constant. There is no requirement on the implementation to follow the signed/unsigned pattern seen for the standard integer types when octal and hexadecimal notation is used for the constants.

integer constant no type

If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type. 841

Commentary

Consider the token 100000000000000000000 in an implementation that supports a 64-bit two's complement **long long**, and no extended integer types. The numeric value of this token outside of the range of any integer type supported by the implementation and therefore it has no type.

This sentence was added by the response to DR #298.

6.4.4.2 Floating constants

```
floating-constant:
                decimal-floating-constant
                hexadecimal-floating-constant
decimal-floating-constant:
                fractional-constant exponent-part_opt floating-suffix_opt
                digit-sequence exponent-part floating-suffix_opt
hexadecimal-floating-constant:
                hexadecimal-prefix hexadecimal-fractional-constant
                                binary-exponent-part floating-suffix_opt
                hexadecimal-prefix hexadecimal-digit-sequence
                                binary-exponent-part floating-suffix_opt
fractional-constant:
                digit-sequence_opt . digit-sequence
                digit-sequence .
exponent-part:
                e sign_opt digit-sequence
                E sign_opt digit-sequence
sign: one of
                +  -
digit-sequence:
                digit
                digit-sequence digit
hexadecimal-fractional-constant:
                hexadecimal-digit-sequence_opt .
                                hexadecimal-digit-sequence
                hexadecimal-digit-sequence .
binary-exponent-part:
                p sign_opt digit-sequence
                P sign_opt digit-sequence
hexadecimal-digit-sequence:
                hexadecimal-digit
                hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
                f  l  F  L
```

Commentary

The majority of *floating-decimal-constant*s do not have an exact binary representation. For instance, if FLOAT_RADIX is 2 then only 4% of constants having two digits after the decimal point can be represented exactly (i.e., those ending .00, .25, .50, and .75).

Unlike an *integer-suffixe*, a *floating-suffix* specifies the actual type, not the lowest rank of a set

of types (not that floating-point types have rank).

Hexadecimal floating constants were introduced to remove the problems associated with translators incorrectly mapping character sequences denoting decimal floating constants to the internal representation of floating numbers used at execution time. The potential mapping problems only apply to the significand, so a decimal representation can still be used for the exponent (requiring a hexadecimal representation for the exponent would have made it harder for human readers to quickly gauge the magnitude of a constant and created a lexical ambiguity, e.g., would the character sequence `p0x1f` be interpreted as ending in the *floating-suffix* `f` or not).

The exponent is always required for the hexadecimal notation, unlike decimal floating constants, otherwise, the translator would not be able to resolve the ambiguity that occurs when a `f`, or `F`, appears as the last character of a preprocessing token. For instance, `0x1.f` could mean `1.0f` (the `f` interpreted as a suffix indicating the type float) or `1.9375` (the `f` being interpreted as part of the significand value).

The *hexadecimal-floating-constant* `0x1.FFFFFEp128f` does not represent the IEC 60559 single-format NaN. It overflows to an infinity in the single format.

C90

Support for *hexadecimal-floating-constant* is new in C99. The terminal *decimal-floating-constant* is new in C99 and its right-hand side appeared on the right of *floating-constant* in the C90 Standard.

C++

The C++ syntax is identical to that given in the C90 Standard.

Support for *hexadecimal-floating-constant* is not available in C++.

Other Languages

Support for *hexadecimal-floating-constant* is unique to C. Fortran 90 supports the use of a *KIND* specifier as part of the floating constant. Fortran also supports the use of the letter *D*, rather than *E*, in the exponent part to indicate that the constant has type **`double`** (rather than real, the single-precision default type). Java supports the optional suffixes *f* (type **`float`**, the default) and *d* (type **`double`**)

Coding Guidelines

Mapping to and from a hexadecimal floating constant, and its value as a floating-point literal, requires knowledge of the underlying representation. The purpose of supporting the hexadecimal floating constant notation is to allow developers to remove uncertainty over the accuracy of the mapping, of values expressed in decimal, performed by translators. Developers are unlikely to want to express floating constants in hexadecimal notation for any other reason and the guideline recommendation dealing with use of representation information is not applicable.

representation information using 569.1

Dev 569.1 Floating constant may be expressed using the hexadecimal floating-point notation.

The advantage of hexadecimal floating constants is that they guarantee an exact (when `FLT_RADIX` is a power of two) floating value in the program image, provided the constant has the same or less precision than the type.

integer constant not in visible source 825.3

For the same rationale as integer constants, there is good reason why most floating constants should not appear in the visible source.

Cg 842.1 No floating constant, other than `0.0` and `1.0`, shall appear in the visible source code other than as the sole preprocessing token in the body of a macro definition.

Usage

exponent 334 integer constant usage 825

Exponent usage information is given elsewhere. Also see elsewhere for a discussion of Benford's law and the first non-zero digit of constants (χ^2 = 1,680 is a very poor fit).

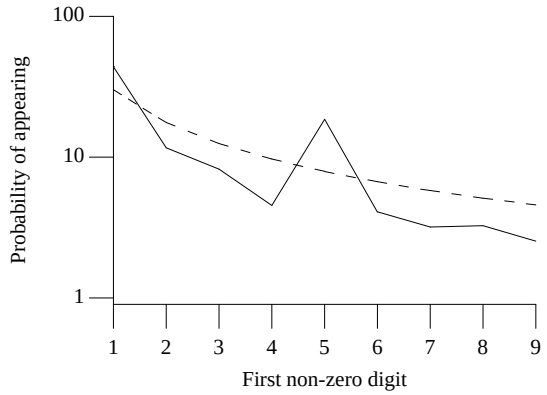


Figure 842.1: Probability of a *decimal-floating-constant* (i.e., not hexadecimal) starting with a particular digit. Based on the visible form of the `.c` files. Dotted line is the probability predicted by Benford's, i.e., $\log(1 + d^{-1})$, where d is the numeric value of the digit.

Table 842.1: Occurrence of various *floating-suffix*es (as a percentage of all such constants). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	`.c` files	`.h` files
none	98.3963	99.7554
F/f	1.4033	0.1896
L/l	0.2005	0.0550

Table 842.2: Common token pairs involving *floating-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, *floating-constant*	0.0	20.4	*floating-constant* /	5.8	1.8
= *floating-constant*	0.1	15.7	*= *floating-constant*	6.3	1.6
* *floating-constant*	0.2	12.5	*floating-constant* *	6.8	0.1
(*floating-constant*	0.0	8.8	*floating-constant* ;	26.5	0.1
+ *floating-constant*	0.4	7.7	*floating-constant*)	25.9	0.1
-v *floating-constant*	0.3	6.7	*floating-constant* ,	25.8	0.1
/ *floating-constant*	2.0	6.4			

Description

843 A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. significand part

Commentary

This defines the terms *significand part* and *exponent part*.

844 The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (`.`), followed by a digit sequence representing the fraction part. whole-number part, fraction part

Commentary

A restatement of information given in the Syntax clause. The character denoting the period, which may appear when floating-point values are converted to strings, is locale dependent. However, the period character that appears in C source is not locale dependent.

A leading zero does not indicate an octal floating-point value.

C++

2.13.3p1 *The integer part, the optional decimal point and the optional fraction part form the significant part of the floating literal.*

The use of the term *significant* may be a typo. This term does not appear in the C++ Standard and it is only used in this context in one paragraph.

Other Languages

This form of notation is common to all languages that support floating constants, although in some languages the period (decimal point) in a floating constant is not optional.

Coding Guidelines

The term *whole-number* is sometimes used by developers. A more commonly used term is *integer part* (the term used by the C++ Standard). The commonly used term for the period character in a floating constant is *decimal point*.

floating constant digit layout

A common mathematical convention is to have a single nonzero digit preceding the period. This is a useful convention when reading source code since it enables a quick estimate of the magnitude of the value to be made. There are also circumstances where more than one digit before the period, or leading zeros before and after the period, can improve readability when the floating constant is one of many entries in a table. In this case the relative position of the first non zero digit may provide a useful guide to the relative value of a series of constants, which may be more important information than their magnitudes.

Your author knows of no research showing that any method of displaying floating constants minimizes the cognitive effort, or the error rate, in comprehending them. However, there does appear to be an advantage in having consistency of visual form between constants close to each other in the source. Comprehending the relationship between the various initializers appears to require less effort for `g_1` and `g_2` than it does for `g_3`.

```
double g_1[] = {
                 1.2,
                 0.12,
                 0.012,
                 0.0012,
                 0.00012,
                };
double g_2[] = {
                 1.2e-0,
                 1.2e-1,
                 1.2e-2,
                 1.2e-3,
                 1.2e-4,
                };
double g_3[] = {
                 1.2e-0,
                 0.12,
                 1.2e-2,
                 0.0012,
                 1.2e-4,
                };
```

floating constant assumed accuracy

Trailing zeros in the fractional part of a floating constant may not affect its value (unless a translator has poor character to binary conversion), but they do contain information. Trailing zeros can be interpreted as a statement of accuracy; for instance, the measurement 7.60 inches is more accurate than 7.6 inches.

Leading zeros are sometimes used for padding and have no alternative interpretation. Adding trailing zeros to a fractional part for padding purposes is misleading. They could be interpreted as giving a floating

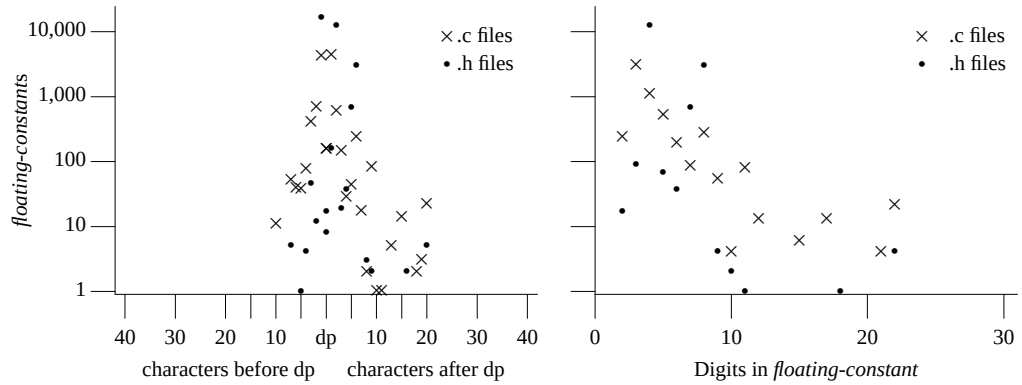


Figure 844.1: Number of *floating-constant*s, that do not contain an exponent part, containing a given number of digit sequences before and after the decimal point (dp), and the total number of digit in a *floating-constant*. Based on the visible form of the `.c` and `.h` files.

constant a degree of accuracy that it does not possess. While such usage does not affect the behavior of a program, it can affect how developers interpret the accuracy of the results.

Rev 844.1 Floating constants shall not contain trailing zeros in their fractional part unless these zeros accurately represent the known value of the quantity being represented.

---

845 The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence.

Commentary

A restatement of information given in the Syntax clause.

C90

Support for *p* and *P* is new in C99.

C++

Like C90, the C++ Standard does not support the use of *p*, or *P*.

Other Languages

The use of the notation *e* or *E* is common to most languages that support the same form of floating constants. Fortran also supports the use of the letter *D*, rather than *E*, to indicate the exponent. In this case the constant has type **double** (there is no type **long double**).

Coding Guidelines

Amongst a string of digits, the letter `E` can easily be mistaken for the digit `8`. There is no such problem with the letter `e`, which also adds a distinguishing feature to the visual appearance of a floating constant (a change in the height of the characters denoting the constant). However, there is no evidence to suggest that this choice of exponent letter is sufficiently important to warrant a guideline recommendation. At the time of this writing there is little experience available for how developers view the exponent `p` and `P`. While the prefix indicates that a hexadecimal constant is being denoted, a lowercase `p` offers an easily distinguished feature that its uppercase equivalent does not.

Example

```
1  double glob[] = {
2                    67E9,
3                    9e76,
4                   };
```

Either the whole-number part or the fraction part has to be present; 846

Commentary

A restatement of information given in the Syntax clause.

Coding Guidelines

When only one of these parts is present, the period character might easily be overlooked, especially when floating constants occur adjacent to other punctuation tokens such as a comma. This problem can be overcome by ensuring that a digit (zero can always be used) appears on either side of the period. However, such usage is not, itself, free of problems. The period can be interpreted as a comma (if the source is being quickly scanned), causing the digits on either side of the period to be treated as two separate constants. The issue of white space between tokens is discussed elsewhere. In the case of digits after the decimal point, there is also the issue of assumed accuracy of the floating constant.

words 770
white space between
floating 844
constant
assumed accuracy

Example

```
1  extern int g(double, int, ...);
2
3  double glob[] = {
4                    12.3, .456, 789.,
5                    12.3,.456,789.,
6                    98.7, 0.654, 321.0,
7                    98.7,0.654,321.0,
8                   };
9
10 void f(void)
11 {
12 g(1.2, 45, 6., 0);
13 g(1.2, 45, 6,0);
14 g(7.8,90, 1.,2);
15 g(3.4,56, 7.0,8);
16 }
```

---

for decimal floating constants, either the period or the exponent part has to be present. 847

Commentary

A restatement of information given in the Syntax clause. Without one of these parts the constant would be interpreted as an integer constant.

Coding Guidelines

Is there a benefit, to readers, of including a period in the visible representation of a floating constant when an exponent part is present? Including a period further differentiates the appearance of a floating constant from that of other types of constants. Developers with a background in numerate subjects will have frequently encountered values that contain decimal points. The exponent notation used in C source code is rarely encountered outside of source code, powers of ten usually being written as just that (e.g., 10^2). Because of these relative practice levels, developers are much more likely to be able to automatically recognize a floating value with a constant that contains a period than one that only contains an exponent (which is likely to require

automatization 0

conscious attention). However, given existing usage (see Figure 844.1) a guideline recommendation does not appear worthwhile.

Developers reading source often only need an approximate estimate of the value of floating constants. The first few digits and the power of ten (sometimes referred to as the *order of magnitude* or simply the *magnitude*) contain sufficient value information. The magnitude can be calculated by knowing the number of nonzero digits before the decimal point and the value of the exponent. There are many ways in which these two quantities can be varied and yet always denote the same value. Is there a way of denoting a floating constant such that its visible appearance minimizes the cognitive effort needed to obtain an estimate of its value? The possible ways of varying the visible appearance of a floating constant including:

- Not using an exponent; the magnitude is obtained by counting the number of digits in the whole-number part.
- Having a fixed number of digits in the whole-number part, usually one; the magnitude is obtained by looking at the value of the exponent.
- Some combination of digits in the whole-number part and the exponent.

There are a number of factors that suggest developers' effort will be minimized when small numbers are written using only a few digits before the decimal point rather than using an exponent, including the following:

- Numbers occur frequently in everyday life and people are practiced at processing the range of values they commonly encounter. The prices of many items in shops in the UK and USA tend to have only a few digits after the decimal point, while in countries such as Japan and Italy they tend to have more digits (because of the relative value of their currency).
- *Subitizing* is the name given to the ability most people have of instantly knowing the number of items in a small set (containing up to five items, although some people can only manage three) without explicitly counting them. 1641 subitizing

Your author does not know of any algorithm that optimizes the format (i.e., how many digits should appear before a decimal point or selecting whether to use an exponent or not) in which floating-point constants appear, such that reader effort in extracting a value from them is minimized.

Example

Your author is not aware of any studies investigating the effect that the characteristics of human information processing (e.g., the Stroop effect) have on the probability of the value of a constant being misinterpreted. 1641 stroop effect

```
1  double d[] = {
2                 123.567,
3                 01.1,
4                 3.333e2,
5                 1.23456e8,
6                 1111.3,
7                 122.12e2,
8                };
```

Semantics

848 The significand part is interpreted as a (decimal or hexadecimal) rational number;

Commentary

One form is based on the human, base 10, representation of values, the other on the computer, base 2, representation.

C90

Support for hexadecimal significands is new in C99.

C++

The C++ Standard does not support hexadecimal significands, which are new in C99.

Other Languages

While support for the hexadecimal representation of floating constants may not be defined in other language standards, some implementations of these languages (e.g., Fortran) support it.

Example

What is the IEC 60559 single-precision representation of `12.345`? For the digits before the decimal point we have:

$$12_{10} = 1100_2 \tag{848.1}$$

For the digits after the decimal point we have:

```
   .345
   ×  2
0  .690
   ×  2
1  .380
   ×  2
0  .760
   ×  2
1  .520
   ×  2
1  .040
   ×  2
0  .080
   ×  2
0  .160
...
```

$$.345_{10} = .01000101100001010001111 1_2 \tag{848.2}$$

Writing the number in normalized form, we get:

$$1100.01011000010100011111 \times 2^0 = 1.10001011000010100011111 \times 2^3 \tag{848.3}$$

Representing the number in single-precision, the exponent bias is 127, giving an exponent of $127 + 3 = 130_{10} = 10000010_2$. The final bit pattern is (where | indicates the division of the 32-bit representation into sign bit, exponent, and significand):

$$0 \mid 10000010 \mid 10001011000010100011111 \tag{848.4}$$

What is the decimal representation of the hexadecimal floating-point constant, assuming an IEC 60559 representation of `0x0.12345p0`? For the significand we have:

$$.12345_{16} = .0001001000011010001012 = 1.0010001101000101 \times 2^{-4} \tag{848.5}$$

For the exponent we have:

$$127 - 4 = 123_{10} = 1111011_2 \tag{848.6}$$

which gives a bit pattern of:

$$0 \mid 1111011 \mid 00100011010001010000000 \tag{848.7}$$

Converting this to decimal, the exponent is $1111011_2 = 123 - 127 = -4_{10}$; and restoring the implicit 1 in the significand, we get the value:

$$1.0010001101000101_2 \times 2^{-4} \tag{848.8}$$
$$0.0001001000011010001012 \tag{848.9}$$
$$(1/16 + 1/128 + 1/2048 + 1/4096 + 1/16384 + 1/262144 + 1/1048576)_{10} \tag{848.10}$$
$$(74565/1048576)_{10} \tag{848.11}$$
$$7.111072540283203125\ldots_{10} \times 10^{-2} \tag{848.12}$$

Taking into account the accuracy of the representation, we get the value $7.111073_{10} \times 10^{-2}$.

---

849 the digit sequence in the exponent part is interpreted as a decimal integer.

Commentary

Even if the significand is in hexadecimal notation, the exponent is still interpreted as a decimal quantity.

C++

. . . , an optionally signed integer exponent, . . . 2.13.3p1

There is no requirement that this integer exponent be interpreted as a decimal integer. Although there is wording specifying that both the integer and fraction parts are in base 10, there is no such wording for the exponent part. It would be surprising if the C++ Standard were to interpret 1.2e011 as representing 1.2×10^9; therefore this issue is not specified as a difference.

---

850 For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled.

Commentary

This specification is consistent with that for the significand.

---

851 For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled.

Commentary

Scaling the significand of a hexadecimal floating constant by a power of 2 means that no accuracy is lost when all the powers of 2 specified by the exponent are representable using the value of FLT_RADIX. (This is always true when FLT_RADIX has a value of 2, as specified by IEC 60559.) This scaling is performed during translation; it is not an execution-time issue.

C90

Support for hexadecimal floating constants is new in C99.

C++

The C++ Standard does not support hexadecimal floating constants.

---

floating constant representable value chosen

For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. 852

Commentary

int to float 689 nearest representable value

This behavior is different from that specified for integer-to-float conversion. Having introduced hexadecimal floating constants, the Committee could not then restrict their use to implementations having a FLT_RADIX that was a power of 2. The advantage of exact representation does not always occur in implementations where FLT_RADIX is not a power of 2, but a translator is still required to support this form of floating constant.

hexadecimal constant 859 not represented exactly

When converting a value to a different base the best approximation can be obtained using a finite automaton if the two bases are both an integral power of a common integral root.[247] For hexadecimal floating constants this requires that the other base be a power of 2. Unless two bases have this property it is not possible to always find the best approximation, when converting a value between them, using a finite automaton (there are some numbers that require arbitrary precision arithmetic to obtain the best approximation). Thus, it is not possible to find the best n-bit binary approximation of a decimal real number using a finite automaton.[247] For this reason the C Standard does not require the best approximation and will accept the nearest representable values either side of the best approximation (see Figure 852.1).

Floating constants whose values have the same mathematical value, but are denoted by different character sequences (e.g., 1.57 and 15.7e-1), may be mapped to different representable values by a translator. (Depending on how its mapping algorithm works, the standard permits three different possibilities.) In fact the standard does not even require that implementations map the same sequence of characters to the same value internally (although an implementation that exhibited such behavior would probably be considered to have low quality). Neither is there any requirement, in this case, that if the constant value is exactly representable the exact representation be used.

DECIMAL_DIG conversion 379 recommended practice

C90

Support for hexadecimal floating constants is new in C99.

C++

2.13.3p1 *If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.*

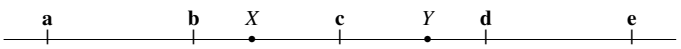


Figure 852.1: The nearest representable value to X is b, however, its value may also be rounded to a or c. In the case of Y, while d is the nearest representable value the result may be rounded to c or e.

Other Languages

This issue is not C specific. All languages have to face the problem that only a finite number of floating-point values can be represented and the mapping from a string to a binary representation may not be exact.

Common Implementations

A few implementations continue to use the *student* solution of multiply and add, a character at a time, moving left-to-right through the significand of the floating constant, followed by a loop that multiplies, or divides, the result by 10; the iteration count being given by the value of the exponent. Your author knows of no translator that maps identical floating constant tokens into different internal representations. Mapping different floating constant tokens, which have the same mathematical value, to different internal representations is not unknown.

Coding Guidelines

There are a number of factors that can introduce errors into the conversion of floating-point tokens (a sequence of characters) into the internal representation used for floating values (a pattern of bits) by the translator, including:

- Poor choice of conversion algorithm by the translator vendor, leading to larger-than-permitted, by the standard, error
- Poor choice of rounding direction when performing conversions
- The finite set of numbers that can be represented in any model of floating-point numbers

Translator errors in conversion of floating constants are most noticeable, by developers, when they have an exact representation in an integer type. In:

```
#include <stdio.h>

void f(void)
{
double d = 6.0;

if ((int)d != 6)
   printf("Oh dear\n");
/* ... */
}
```

The standard does not require that the character sequence `6.0` be converted to the floating value 6.0. The value 5.9999999 (continuing for a suitable number of 9s) is the smaller representable value immediately adjacent to the nearest representable value— a legitimate result— which, when cast to **`int`**, yields a value of 5. A translator that chooses to round-to-even, or not round toward zero, would not exhibit this problem.

These coding guidelines deal with developer-written code. Developers do not need to be told, in a guideline, to use high-quality implementation. If developers are in the position of having to use an implementation that does not correctly convert floating constant tokens, then all that can be suggested is that hexadecimal floating constants be used. The error introduced by the model used by an implementation to represent floating numbers has to be lived with. The issue of converting floating values to integer types is discussed elsewhere.

686.1 floating constant converted exactly

Example

```
#include <stdio.h>

int main(void)
{
if ((1.57 != 1.570) || (1.57 != 0.157E1) ||
    (0.0 != 0.000) || (0.0 != 0.0E10))
   printf("You can never enter the same stream twice\n");
}
```

For hexadecimal floating constants when **FLT_RADIX** is a power of 2, the result is correctly rounded. 853

Commentary

If FLT_RADIX is a power of 2, the hexadecimal value has a unique mapping to the representation of the significand (any additional digits, in excess of those required by the type of the floating constant, are used to form the correctly rounded result).

correctly rounded result 64

Common Implementations

All known implementations have a FLT_RADIX that is a power of 2.

An unsuffixed floating constant has type **double**. 854

Commentary

A floating constant has type **double**, independent of its value. This situation differs from the integer types where the value of the literal (as well as its form) is used to determine its type. There is no suffix to explicitly denote the type **double**.

Other Languages

Most languages do not have more than one floating-point type. A literal could only have a single type in these cases. A floating constant in Fortran has type **real** by default.

Common Implementations

Some translators[55] provide an option that allows this default behavior to be changed (perhaps to improve performance by using **float** rather than **double**).

Coding Guidelines

If an unsuffixed floating constant is immediately converted to another floating type, would it have been better to use a suffix? An immediate conversion of a floating constant to another floating type suggests that the developer is being sloppy and is omitting the appropriate suffix. In the case of a conversion to type **long double**, there is also the possibility of a loss of precision. The floating constant token is converted to **double** first, potentially losing any additional accuracy present in the original token, before being converted to **long double**; however, the issues are not that simple.

The precision to which a floating constant is represented internally by a translator need not be related to its type. It can be represented to less or greater precision depending on the value of the FLT_EVAL_METHOD macro. A suffix changes the type of a floating constant, but need not change the precision of its internal representation. On the other hand, a cast operation is required to remove any additional precision in a value, but does not have the information needed to provide additional precision.

FLT_EVAL_METHOD 354

Explicitly casting floating constants ensures the result is consistent (assuming the cast occurs during program execution, not at translation time, which can only happen if the FENV_ACCESS pragma is in the ON state) with casts applied to nonconstant operands having floating type.

The disadvantage of specifying a suffix on a floating constant, because of the context in which the constant is used, is that the applicable type may change. The issues involved with implicit conversion versus explicit conversion are discussed elsewhere. An explicit cast, using a typedef name rather than a suffix, is more flexible in this regard.

implicit conversion 654
integer constant with suffix, not immediately converted 835.2

The following guideline recommendations mirror those given for suffixed integer constants, except that it is specified as a review item, not a coding guideline. The process of resolving whether a suffix or cast is best in light of possible settings for the FLT_EVAL_METHOD needs human attention.

Rev 854.1 A floating constant containing a suffix shall not be immediately converted to another type.

Dev 854.1 The use of a macro defined in a system header may be immediately converted to another type.

Dev 854.1 The use of a macro defined in a developer-written system header may be immediately converted to another type, independent of how the macro is implemented.

Dev 854.1 The body of a macro may convert, to a floating type, one of the parameters of that macro definition.

Example

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  if ((1.0f / 3.0f) != ((float)1.0 / (float)3.0))
6     printf("No surprises here (float)\n");
7  if ((1.0 / 3.0) != ((double)1.0 / (double)3.0))
8     printf("No surprises here (double)\n");
9  if ((1.0L / 3.0L) != ((long double)1.0 / (long double)3.0))
10    printf("No surprises here (long double)\n");
11 }
```

---

855 If suffixed by the letter **f** or **F**, it has type **float**.

Commentary

Unlike its integer counterpart, **short**, there is a suffix for denoting a floating constant of type **float**. Hexadecimal floating constants require the *p* so that any trailing *f* is interpreted as the type **float** suffix rather than another hexadecimal digit.

Other Languages

Java supports the suffixes *f*, or *F*, to indicate type **float** (the default type for floating constants).

Coding Guidelines

What is the developers' intent when giving a floating constant a suffix denoting that it has type **float**? The type of a floating constant may not affect the evaluation type of an expression in which it is an operand (which is controlled by the value of the FLT_EVAL_METHOD macro). Even the value of the floating constant itself may be held to greater precision than the type **float**. [354] FLT_EVAL_ME

---

856 If suffixed by the letter **l** or **L**, it has type **long double**.

Commentary

If greater precision, or range of exponent than type **double**, is required in a floating constant, using the type **long double** suffix may offer a solution. However, possible interaction value of the FLT_EVAL_METHOD macro needs to be taken into account. [354] FLT_EVAL_ME

Coding Guidelines

Using the l suffix in a floating constant may lead to confusion with the digit 1.

Cg 856.1 If a *floating-suffix* is required only the forms *F* or *L* shall be used.

---

857 Floating constants are converted to internal format as if at translation-time.

floating constant internal format

Commentary

The key phrase here is *as if*. Translators wishing to maintain a degree of host independence can still translate to some intermediate form (which is converted to host-specific format as late as program startup). However, all the implications of the conversion must occur at translation-time, not during program execution or startup.

C90

No such requirement is explicitly specified in the C90 Standard.

In C99 floating constants may convert to more range and precision than is indicated by their type; that is, `0.1f` may be represented as if it had been written `0.1L`.

FLT_EVAL_METHOD 354

C++

Like C90, there is no such requirement in the C++ Standard.

Common Implementations

There are a few implementations that use an intermediate form to represent floating constants, independent of the final host on which a program image will execute. The intent is to create a degree of software portability at a lower level than the source code; for instance, implementations that interpret source code that has been translated into the instructions of some abstract machine.

---

floating constant conversion not raise exception

The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. 858

Commentary

This is a requirement on the implementation. A floating constant that is outside the representable range of floating-point values cannot cause an execution-time exception to be raised. Whether a translator issues a diagnostic if it encounters a floating constant that, had it occurred during program execution, would have raised an exception is a quality-of-implementation issue.

C90

No such requirement was explicitly specified in the C90 Standard.

C++

Like C90, there is no such requirement in the C++ Standard.

Recommended practice

---

hexadecimal constant not represented exactly

The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; 859

Commentary

floating constant representable value chosen 852

The issue of exact representation of floating constants is not unique to hexadecimal notation, but the intent of this notation could lead developers to expect that an exact representation will always be used. An inexact representation can occur if `FLT_RADIX` macro has a value that is not a power of 2, or the hexadecimal floating constant contains more digits than are representable in the significand used by the implementation for that floating type This is a quality-of-implementation issue, one of the many constructs that a quality implementation might be expected to diagnose. However, the floating-point contingent on the Committee is strong and hexadecimal constants are a new construct, so we have a recommended practice.

FLT_RADIX 366

C90

Recommended practices are new in C99, as are hexadecimal floating constants.

C++

The C++ Standard does not specify support for hexadecimal floating constants.

Coding Guidelines

A hexadecimal floating constant is a notation intended to be used to provide a mechanism for exactly representing floating-point values. A source file may contain a constant that is not exactly representable. However, support for hexadecimal floating constant notation is new in C99 and at the time of this writing insufficient experience on their use is available to know if any guideline recommendation is worthwhile.

Example

```
1   float f = 0x1.11111111111111111111111111111111111111111111111111111111111111p11;
```

---

860 the implementation should then proceed with the translation of the program.

Commentary

Thanks very much.

---

861 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as **strtod**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.[64)]

Commentary

The C library contains many of the support functions needed to write a translator. Making these functions available in the C library is just as much about allowing implementation vendors to recycle their code as allowing behaviors to be duplicated. If the translator and its library both use identical functions for performing numeric conversions, there is likely to be consistent behavior between the two environments. The function `strtod` is an example of one such case. However, its behavior will only be the same if the various floating-point mode flags are also the same in both environments.

C90

This recommendation is new in C99.

C++

No such requirement is explicitly specified in the C++ Standard.

Other Languages

A common characteristic of many language translators is that they are written in the language they translate, even Cobol. If a translator is not written in its own language, the most common implementation language is C. Languages invariably specify functionality that performs conversions between character strings and floating-point values. However, it is their I/O mechanisms that perform this conversion and the generated character strings, or floating-point values, are not always otherwise available to an executing program.

Fortran has a large number of numeric functions in its library as does Java. Both of these languages have an established practice of writing their libraries in their respective languages. Although in the case of Fortran, there is also a history of using functions written in C. Java defines conversion functions in `java.lang.float` and `java.lang.double`.

Common Implementations

The majority of C translators are written in C and call the identical functions to those provided in their runtime library.

Coding Guidelines

Floating-point values can appear during program execution through two possible routes. They can be part of the character sequence that is translated to form the program image, or they can be read in as character sequences from a stream and converted using the functions defined in Clause 7.20.1.3. In a freestanding

environment the host processor floating-point support is likely to be different (it may be implemented via calls to internal implementation library functions) from that available during translation.

It is possible for the original source of the floating-point numbers to be the same; for instance, a file containing a comma separated list of values and this file being **#include**d at translation time and read during program execution.

Rev 861.1

A program shall not depend on the value of a floating constant appearing in the source code being equal to the value returned by a call to `strtod` with the same sequence of characters as its first argument.

Example

```
#include <stdlib.h>
#include <stdio.h>

#define flt_constant_eq_flt_string(x) (x == strtod(#x, NULL))

void f(void)
{
if (!flt_constant_eq_flt_string(1.2345))
   printf("This translator may not be implemented using its own library\n");
}
```

footnote 64

64) The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.20.1.3). 862

Commentary

Clause 7.20.1.3 describes the `strtod`, `strtof`, and `strtold` functions.

C++

There observation is not made in the C++ Standard. The C++ Standard includes the C library by reference, so by implication this statement is also true in C++.

6.4.4.3 Enumeration constants

enumeration-constant: 863
identifier

Commentary

There is no phase of translation where an identifier that is an *enumeration-constant* is replaced by its constant value.

C++

The C++ syntax uses the terminator *enumerator*.

Other Languages

Other languages that contain enumeration constants also specify them as identifiers.

Coding Guidelines

enumeration constant naming conventions 792

The issue of naming conventions for enumeration constant identifiers is discussed elsewhere.

Semantics

enumeration constant type

An identifier declared as an enumeration constant has type **int**. 864

Commentary

There is no requirement that the enumerated type containing this enumeration constant also have type **int**, although any constant expression used to specify the value of an enumeration constant is required to be representable in an **int**.

1440 enumeration constant representable in int

C++

Following the closing brace of an enum-specifier, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value. If an initializer is specified for an enumerator, the initializing value has the same type as the expression. If no initializer is specified for the first enumerator, the type is an unspecified integral type. Otherwise the type is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.

7.2p4

This is quite a complex set of rules. The most interesting consequence of them is that each enumerator, in the same type definition, can have a different type (at least while the enumeration is being defined):
In C the type of the enumerator is always **int**. In C++ it can vary, on an enumerator by enumerator basis, for the same type definition. This behavior difference is only visible outside of the definition if an initializing value is calculated by applying the **sizeof** operator to a prior enumerator in the current definition.

```
#include <limits.h>

enum TAG { E1 = 2L,             // E1 has type long
           E2 = sizeof(E1),     // E2 has type size_t, value sizeof(long)
           E3 = 9,              // E3 has type int
           E4 = '4',            // E4 has type char
           E5 = INT_MAX,        // E5 has type int
           E6,                  // is E6 an unsigned int, or a long?
           E7 = sizeof(E4),     // E2 has type size_t, value sizeof(char)
         }                      // final type is decided when the } is encountered
           e_val;

int probably_a_C_translator(void)
{
return (E2 == E7);
}
```

Source developed using a C++ translator may contain enumeration with values that would cause a constraint violation if processed by a C translator.

```
#include <limits.h>

enum TAG { E1 = LONG_MAX }; /* Constraint violation if LONG_MAX != INT_MAX */
```

Other Languages

In most other languages that contain enumeration constants, the type of the enumeration constant is the enumerated type, a type that is usually different from the integer types.

Common Implementations

Some implementations support the use of the type **short** and **long**, in the enumeration declaration,[1040] to explicitly specify the type of the enumeration constant, while others[927] use pragmas, or command line options.[353]

Coding Guidelines

enumeration 517 set of named constants

The potential uses for enumeration constants are discussed elsewhere. On the whole these uses imply that an enumeration constant belongs to a unique type— the enumerated type it is defined within. Treating an enumeration constant as having some integer type only becomes necessary when use is made of its value—for instance, as an operand in an arithmetic or bitwise operation. The issues involved in mixing objects having enumerated type and the associated enumeration constants of that type as operands in an expressions are discussed elsewhere.

enumeration 517 set of named constants

Forward references: enumeration specifiers (6.7.2.2). 865

6.4.4.4 Character constants

---

charac-ter constant syntax
escape sequence syntax

866

```
character-constant:
                ' c-char-sequence '
                L' c-char-sequence '
c-char-sequence:
                c-char
                c-char-sequence c-char
c-char:
                any member of the source character set except
                       the single-quote ', backslash \, or new-line character
                escape-sequence
escape-sequence:
                simple-escape-sequence
                octal-escape-sequence
                hexadecimal-escape-sequence
                universal-character-name
simple-escape-sequence: one of
                \'  \"  \?  \\
                \a  \b  \f  \n  \r  \t  \v
octal-escape-sequence:
                \ octal-digit
                \ octal-digit octal-digit
                \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
                \x hexadecimal-digit
                hexadecimal-escape-sequence hexadecimal-digit
```

Commentary

character 776 ' or " matches

The following is an undefined behavior potentially leading to a syntax violation:

```
1   char c = '\z';
```

footnote 879 65

because the characters that may follow a backslash are enumerated by the syntax. A backslash must be followed by one of the characters listed in the C sentence. This character sequence tokenizes as:

```
{char} {c} {=} {'}{\}{z}{'}{;}
```

character 886 constant single character value

A character-constant, that does not contain more than one character, is effectively preceded by an implicit conversion to the type **char**. If this type is signed and the constant sufficiently large, the resulting value is likely to be negative (the most common, undefined behavior, being to wrap).

Rationale

Proposals to add `'\e'` for ASCII ESC (`'\033'`) were not adopted because other popular character sets have no obvious equivalent.

Including *universal-character-name* as an *escape-sequence* removes the need to explicitly include it in other wording that refers to escape sequences (e.g., in conditional inclusion).

1881 #if escape sequences
2037 escape sequences future language directions

Lowercase letters as escape sequences are also reserved for future standardization, although permission is given to use other letters in extensions.

C90

Support for *universal-character-name* is new in C99.

C++

The C++ Standard classifies *universal-character-name* as an *escape-sequence*, not as a *c-char*. This makes no difference in practice to the handling of such *c-char*s.

Other Languages

In other languages the sequence of characters within a character constant usually forms part of the lexical grammar, not the language syntax. The constructs supported by *simple-escape-sequence* was once unique to C. A growing number of subsequent language (e.g., C++, C#, and Java; although Java does not support \?, \a, \v, or hexadecimal escape sequences). Perl also includes support for the escape sequences \e (escape) and \cC (Control-C).

Common Implementations

While lexing a character constant most translators simply look for a closing single-quote to match the one that began the *character-constant*. The internal, syntactic structure of the characters constant is not usually examined until a later phase of translation. As a consequence, many implementations do not diagnose the previous example as a violation of syntax.

Coding Guidelines

Character constants are usually used in code that processes data consisting of sequences of characters. This data may be, for instance, a line of text read from a file or command options typed as input to an executing program. Character constants are not usually the operands of arithmetic operators (see Table 866.3) or the index of an array (581 instances, or 1.4% of all character constants, versus 90,873 instances, or 5.3% of all integer constants).

825.3 integer constant not in visible source
842.1 floating constant not in visible source
822 constant syntax

Does the guideline recommendation dealing with giving symbolic names to constants unconditionally apply to character constants? The reasons given for why symbolic names should be used in preference to the constant itself include:

- *The value of the constant is likely to be changed during program maintenance.* Experience shows that the characters of interest to programs that process sequences of characters do not change frequently. Programs tend to process sets of characters (e.g., alphabetic characters), as shown by the high percentage of characters used in case labels (see Table 866.3).

- *A more meaningful semantic association is created.* Developers are experienced readers of sequences of characters (text) and they have existing strong semantic associations with the properties of many characters; for instance, it is generally known that the space character is used to separate words. That is, giving the name `word_delimiter` to the character constant `' '` is unlikely to increase the amount of semantic association.

- *The number of cognitive switches a reader has to perform is reduced.* Most of the characters constants used (see Figure 884.1) have printable glyphs. Are more or fewer cognitive switches needed to comprehend that, for instance, either `word_delimiter` or `' '` is a character used to delimit words? Use of the escape sequence `'\x20'` would require readers to make several changes of mental representation.

0 cognitive switch

The distribution of characters constants in the visible source (see Figure 884.1) has a different pattern from that of integer constants (see Figure 825.1). There are also more character constants whose lexical form might be said to provide sufficient symbolic information (e.g., the null character, new-line, space, the digit zero, etc.).

The number of cases where the appearance of a symbolic name, in the visible source, is more cost effective than a character constant would appear to be small. Given the cost of checking all occurrences of character constants for the few where replacement by a symbolic name would provide a benefit, a guideline recommendation is not considered worthwhile.

Table 866.1: Occurrence of various kinds of *character-constant* (as a percentage of all such constants). Based on the visible form of the `.c` files.

Kind of *character-constant*	% of all *character-constant*s
not an escape sequence	76.1
simple-escape-sequence	8.8
octal-escape-sequence	15.1
hexadecimal-escape-sequence	0.0
universal-character-name	0.0

Table 866.2: Occurrence of *escape-sequence*s within *character-constant*s and *string-literal*s (as a percentage of *escape-sequence*s for that kind of token). Based on the visible form of the `.c` files.

Escape Sequence	% of *character-constant* Escape Sequences	% of *string-literal* escape sequences	Escape sequence	% of *character-constant* Escape Sequences	% of *string-literal* Escape Sequences
\n	18.10	79.15	\b	0.66	0.04
\t	3.90	11.62	\'	3.24	0.02
\"	1.29	3.08	\%	0.00	0.02
\0	52.70	2.06	\v	0.31	0.01
\x	0.12	1.10	\p	0.00	0.01
\2	2.73	1.01	\f	0.44	0.01
\\	5.70	0.61	\?	0.01	0.01
\r	3.01	0.46	\e	0.00	0.00
\3	4.95	0.42	\a	0.11	0.00
\1	2.72	0.35			

Table 866.3: Common token pairs involving *character-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence Second Token
== *character-constant*	7.1	22.8	*character-constant* ||	4.2	4.2
, *character-constant*	0.3	18.1	*character-constant* &&	5.3	3.3
case *character-constant*	8.5	16.7	<= *character-constant*	7.1	1.7
= *character-constant*	0.8	14.2	>= *character-constant*	3.6	1.5
!= *character-constant*	5.3	8.4	*character-constant*)	33.0	0.7
(*character-constant*	0.1	6.1	*character-constant* ,	17.6	0.3
character-constant :	16.7	6.0	*character-constant* ;	16.6	0.3

Description

integer character constant

An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`.

Commentary

The syntax specification does not use the term *integer character constant*. It is used here, and throughout the standard, to distinguish character constants that are not wide character constants. While a character constant has type **int**, the common developer terminology is still *character constant* (the wide form being relatively rare; its use is explicitly called out).

C90

The example of ab as an integer character constant has been removed from the C99 description.

C++

A character literal is one or more characters enclosed in single quotes, as in `'x'`*, . . .* 2.13.2p1

A multibyte character is replaced by a *universal-character-name* in C++ translation phase 1. So, the C++ Standard does not need to refer to such entities here.

Coding Guidelines

The phrase *integer character constant* is rarely heard, although it is descriptive of the type and form of this integer constant. The common usage terminology is *character constant*, which does suggest that the constant has a character type. Coding guideline documents need to use the term *integer character constant* to remind developers that this form of constant has type **int**.

---

868 A wide character constant is the same, except prefixed by the letter **L**.

wide character constant
character constant wide

Commentary

The term *wide character constant* is used by developers to describe this kind of character constant. The prefix is an uppercase *L* only. There is no support for use of a lowercase letter.

Other Languages

Support for some form of wide characters is gradually becoming more generally available in programming languages. Fortran (since the 1991 standard) supports *char-literal-constant*s that contain a prefix denoting the character set used (e.g., `NIHONGO_'some kanji here'`). Later versions of Cobol supported similar functionality.

Common Implementations

While all implementations are required to support this form of character constant, they can vary significantly in the handling of the character sequences appearing within single-quotes. Many implementations simply support the same set of characters, in this context, as when no L prefix is given.

---

869 With a few exceptions detailed later, the elements of the sequence are any members of the source character set;

Commentary

The exceptions can occur through the use of escape sequences. The elements may also denote characters that are not in the source character set if an implementation supports any.

C++

[Note: in translation phase 1, a `universal-character-name` *is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of* `universal-character-name`*s. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained.]* 2.13.2p5

In C++ all elements in the sequence are characters in the source character set after translation phase 1. The creation of *character-literal* preprocessing tokens occurs in translation phase 3, rendering this statement not applicable to C++.

Common Implementations

Most implementations support characters other than members of the source character set. Any character that can be entered into the source code, using an editor, are usually supported within a character constant.

---

character constant mapped

they are mapped in an implementation-defined manner to members of the execution character set. 870

Commentary

translation phase 4 129
translation phase 5 133
translation phase 1 116
footnote 141 1874

This mapping can either occur in translation phase 4 (when the character constant occurs in an expression within a preprocessing directive) or 5 (all other occurrences). Those preprocessing tokens that are part of a preprocessing directive will have the values given them in translation phase 1. The two mapped values need not be the same.

C++

2.13.2p1 *An ordinary character literal that contains a single* `c-char` *has type* **`char`***, with value equal to the numerical value of the encoding of the* `c-char` *in the execution character set.*

2.2p3 *The values of the members of the execution character sets are implementation-defined, . . .*

2.13.2p2 *The value of a wide-character literal containing a single* `c-char` *has value equal to the numerical value of the encoding of the* `c-char` *in the execution wide-character set.*

Taken together, these statements have the same meaning as the C specification.

---

escape sequences
character constant
character constant escape sequences

The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences: 871

```
single quote   '       \'
double quote   "       \"
question mark  ?       \?
backslash      \       \\
octal character        \octal digits
hexadecimal character  \xhexadecimal digits
```

Commentary

The standard defines a set of escape sequences that can occur both in character constants and string literals. It does not define a separate set for each kind of constant. A consequence of this single set is that there are multiple representations for some characters, for the two kinds of constants. Being able to prefix both the `'` and `"` characters with a backslash, without needing to know what delimiter they appear between, simplifies the automatic generation of C source code.

The single-quote `'` and the double-quote `"` characters have special meaning within character constants and string literals. The question-mark `?` character can have special meaning as part of a trigraph. To denote these characters in source code, some form of escape sequence is required, which in turn adds the escape sequence

character (the backslash \ character) to the list of special characters. Preceding any of these special characters with a backslash character is the convention used to indicate that the characters represent themselves, not their special meaning.

Octal and hexadecimal escape sequences provide a mechanism for developers to specify the numeric value of individual execution-time characters within an integer character constant. While the syntax does permit an arbitrary number of digits to occur in a hexadecimal escape sequence, the range of values of an integer character constant cannot be arbitrarily large.

882 escape sequence value within range

The conversion of these escape sequences occurs in translation phase 5.

133 translation phase 5

C++

The C++ wording, in Clause 2.13.2p3, does discuss arbitrary integer values and the associated Table 5 includes all of the defined escape sequences.

Other Languages

A number of languages designed since C have followed its example in offering support for escape sequences. Java does not support the escape sequence \?, or hexadecimal characters.

Common Implementations

Some implementations treat a backslash character followed by any character not in the preceding list as representing the following character only.

Coding Guidelines

Experience has shown that it is easy for readers to be confused by the character sequences \' and \\ when they occur among other integer character constants. However, a guideline recommendation telling developers to be careful serves no useful purpose.

What purpose does having a character constant containing an octal or hexadecimal escape sequence in the source serve? Since the type of an integer character constant is **int**, not **char**, an integer constant with the same value could appear in every context that an integer character constant could appear, with no loss of character set independence (because there was none in the first place).

One reason for using quotes, even around a hexadecimal or octal escape sequence, is to maintain the semantic associations, in a readers head, of dealing with a character value. Although integer character constants have type **int**, experience suggests that developers tend to think of them in terms of a character type. When comprehending code dealing with objects having character types, an initializer, for instance, containing other character constants, or values having semantic associations with character sets, a reader's current frame of mind is likely to be character based not integer based. The presence of a value between single-quotes is perhaps sufficient to maintain a frame of mind associated with character, not integer, values (i.e., there are no costs associated with a cognitive switch).

0 cognitive switch

Another reason for use of an escape sequence in a character constant is portability to C++, where the type of an integer character constant is **char**, not **int**. In this case use of an integer constant would not be equivalent (particularly if the constant was an argument to an overloaded function).

The character sequence \? is needed when sequences of the ? character occur together and a trigraph is not intended.

232 trigraph sequences replaced by

Example

```
#include <stdio.h>

#define TRUE  ('/'/'/')
#define FALSE ('-'-'-')

int main(void)
{
printf("%d%s\n", '\',',"'); // confusing ");
}
```

The double-quote " and question-mark ? are representable either by themselves or by the escape sequences \" and \?, respectively, but the single-quote ' and the backslash \ shall be represented, respectively, by the escape sequences \' and \\. 872

Commentary

The sequence \\ is only mapped once; that is, the sequence \\\\ represents the two characters \\, not the single character \.

Other Languages

To represent the character constant delimiter Pascal uses the convention of two delimiters immediately adjacent to each other (e.g., "'" represents the character single-quote).

escape sequence octal digits

The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. 873

Commentary

The value of the octal digits is taken as representing the value of a mapped single character in the execution character set. It is not possible to use an octal escape sequence to represent a character whose value is greater than what can be represented within three octal digits (e.g., 511). For the most common case of CHAR_BIT having a value of eight, this is not a significant limitation.

CHAR_BIT 307 macro

escape sequence octal value

The numerical value of the octal integer so formed specifies the value of the desired character or wide character. 874

Commentary

The maximum value that can be represented by three octal digits is 511. There are no prohibitions on using any of the octal values within the range of values that can be represented (unlike use of the \u form).

universal 815 character name syntax

Common Implementations

Most implementations use an 8-bit byte, well within the representable range of an octal escape sequence.

Coding Guidelines

character 871 constant escape sequences

As pointed out elsewhere, a character constant is likely to be more closely associated in a developer's mind with characters rather than integer values. One reason for representing characters in the execution character set using escape sequences is that a single character representation may not be available in the source character set. Is there any reason for a character constant to contain an octal escape sequence whose value represents a member of the execution character set that is representable in the source character set? There are two possible reasons:

1. For wide character constants, the mapping to the execution character might not be straight-forward. Use of escape sequences may be the natural notation to use to represent members. In this case it is possible that an escape sequence just happens to have the same value as a character in the source character set.
2. The available keyboards may not always support the required character (trigraphs are only available for characters in the basic source character set).

basic source 221 character set

Example

```
1  char CHAR_EOT = '\004';
2  char CHAR_DOL = '\044';
3  char CHAR_A   = '\101';
```

875 The hexadecimal digits that follow the backslash and the letter **x** in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant.

escape sequence hexadecimal digits

Commentary

A hexadecimal escape sequence can denote a value that is outside the range of values representable in an implementation's **char** or `wchar_t` type. However, the specification states that an escape sequence represents a single character. This means that a large value cannot be taken to represent two or more characters (in an implementation that supports more than one character in an integer character constant) by using a hexadecimal value that represents the combined value of those two mapped characters.

Coding Guidelines

Occurrences of hexadecimal escape sequences are very rare in the visible source of the `.c` files. Given this rarity, developers are much more likely to be unfamiliar with the range of possible behaviors of hexadecimal escape sequences than the behaviors of octal escape sequences. Possible unexpected behaviors include:

866 escape sequence syntax
873 escape sequence octal digits

- The character 0 appears before the x or X in a hexadecimal constant, but not between the backslash and the *x* in an escape sequence.
- An arbitrary number of digits are permitted in a hexadecimal escape sequence, unlike octal escape sequences that contain a maximum of three digits. Padding the numeric part of hexadecimal escape sequences with leading zeros can provide a misleading impression. It is possible for a character not intended, by the developer, to be part of the hexadecimal escape sequence to be treated, by a translator, as being part of that sequence. (This requires implementation support for more than one character in a character constant.) This situation is more likely to occur in string literals.

877 escape sequence longest character sequence
885 character constant more than one character
895 string literal syntax

Cg 875.1 A hexadecimal escape sequence shall not be used in an integer character constant.

Example

```
1  char ch_1 = '\x00000000000000000000000041';
2  char ch_2 = '\0x42';
```

876 The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

escape sequence hexadecimal value

Commentary

While there is no restriction on the number of digits in a hexadecimal escape sequence, there is a requirement that the value be in the range representable by the type **unsigned char** (which on some implementations is 32 bits) or the type `wchar_t`. There are no prohibitions on using any of the hexadecimal values within the range of values that can be represented (unlike use of the *\u* form).

882 escape sequence value within range
815 universal character name syntax

Common Implementations

The *desired character* in the execution environment, that is. As far as most translators are concerned, the numerical value is a bit pattern. In general they have no knowledge of which, if any, character this represents in the execution character set.

877 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

escape sequence longest character sequence

Commentary

This specification resolves an ambiguity in the syntax. Although it is a little misleading in that it can be read to suggest that both octal and hexadecimal escape sequences may consist of an arbitrary long sequence of characters, the syntax permits a maximum of three digit characters in an octal escape sequence.

Coding Guidelines

Character constants usually consist of a single character, so much of the following discussion is not going to be applicable to them as often as it is to string literals.

For both forms of escape sequence there is the danger that any applicable digit characters immediately following them will be taken to be part of that sequence. The possible number of digits in an octal escape sequence is limited to three. If this number of digits is always used, there is never the possibility of a following character being included in the sequence.

Cg 877.1 An octal escape sequence shall always contain three digits.

Dev 877.1 The lexical form `'\0'` may be used to represent the null character.

CHAR_BIT 307 macro If `CHAR_BIT` has a value greater than 9, an octal escape sequence will not be able to denote all of the representable values in the type **`unsigned char`**.

Dev 875.1 A hexadecimal escape sequence may be used if an octal escape sequence cannot represent the required value.

Example

```
1  char a_char = '\02';
2  char null_char = '\0';
3  char b_char = '\0012';
```

In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, and **\v**.[65)] 878

Commentary

character display semantics A restatement of information given in the Syntax clause. The characters are nongraphical in the sense that they do not represent a glyph corresponding to a printable character. Their semantics are discussed elsewhere.

C90

Support for universal character names is new in C99.

C++

Apart from the rule of syntax given in Clause 2.13.2 and Table 5, there is no other discussion of these escape sequences in the C++ Standard. :-O

Other Languages

Support for universal character names is slowly being added to other ISO standardized languages. Java contains similar escape sequences.

Common Implementations

Some implementations define the escape sequence `\e` to denote *escape*; the Perkin-Elmer C compiler[1075] used escape sequences to represent characters not always available on keywords of the day.

digraphs 916

879 65) The semantics of these characters were discussed in 5.2.2.

footnote 65

Commentary

This discussion describes the effect of writing these characters to a display device.

character display semantics

880 If any other character follows a backslash, the result is not a token and a diagnostic is required.

Commentary

Such a sequence of characters does not form a valid preprocessing token (e.g., `'\D'` contains the preprocessing tokens {`'`}, {`\`}, {`D`}, and {`'`}, not the preprocessing token {`'\D'`}). When a preprocessing token contains a single-quote, the behavior is undefined.

776 character ' or " matches

776 character ' or " matches

It is possible that an occurrence of such a character sequence will cause a violation of syntax to occur, which will in turn require a diagnostic to be generated. However, implementations are not required to issue a diagnostic just because a footnote (which is not non-normative) says one is required.

C90

If any other escape sequence is encountered, the behavior is undefined.

C++

There is no equivalent sentence in the C++ Standard. However, this footnote is intended to explicitly spell out what the C syntax specifies. The C++ syntax specification is identical, but the implications have not been explicitly called out.

Common Implementations

Most C90 translators do not issue a diagnostic for this violation of syntax. They treat all characters between matching single-quotes as forming an acceptable character constant.

Coding Guidelines

There is existing practice that treats a backslash followed by another character as being just that character (except for the special cases described previously). The amount of source that contains this usage is unknown. Removing the unnecessary backslash is straight-forward for human-written code, but could be almost impossible for automatically generated code (access to the source of the generator is not always easy to obtain). Customer demand will ensure that translators continue to have an option to support existing practice.

881 See "future language directions" (6.11.4).

Constraints

882 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the type **`unsigned char`** for an integer character constant, or the unsigned type corresponding to **`wchar_t`** for a wide character constant.

escape sequence value within range

Commentary

A character can be represented in an object of type **`char`**, and all members of the basic source character set are required to be represented using positive values. This constraint can be thought of as corresponding to the general constraint given for constants being representable within their type. (Although integer character constants have type **`int`** they are generally treated as having a character type.)

477 char hold any member of execution character set

478 basic character set positive if stored in char object

823 constant representable in its type

C++

2.13.2p4

The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for ordinary literals) or **wchar_t** *(for wide literals).*

The wording in the C++ Standard applies to the entire character literal, not to just a single character within it (the C case). In practice this makes no difference because C++ does not provide the option available to C implementations of allowing more than one character in an integer character constant.

The range of values that can be represented in the type **char** may be a subset of those representable in the type **unsigned char**. In some cases defined behavior in C becomes implementation-defined behavior in C++.

```
1  char *p = "\0x80"; /* does not affect the conformance status of the program */
2                     // if CHAR_MAX is 127, behavior is implementation-defined
```

In C a value outside of the representable range causes a diagnostic to be issued. The C++ behavior is implementation-defined in this case. Source developed using a C++ translator may need to be modified before it is acceptable to a C translator.

Common Implementations

Some implementations perform a modulo operation on the value of the escape sequence to ensure it is within the representable range of the type **unsigned char**.

Example

```
1  unsigned char uc_1 = '\xffff';
2    signed char sc_1 = '\xff';
```

Semantics

character constant type

An integer character constant has type **int**. 883

Commentary

A character constant is essentially a way of representing the value of a character in the execution character set in a notation that is translator independent. C does not consider a character constant to be a special case of a string literal of length one (as some other languages do).

C++

2.13.2p1 *An ordinary character literal that contains a single* c-char *has type* **char***, . . .*

The only visible effect of this difference in type, from the C point of view, is the value returned by sizeof. In the C++ case the value is always 1, while in C the value is the same as sizeof(int), which could have the value 1 (for some DSP chips), but for most implementations is greater than 1.

2.13.2p1 *A multicharacter literal has type* **int** *and implementation-defined value.*

The behavior in this case is identical to C.

Other Languages

In most other languages a character constant has a character type.

Coding Guidelines

A common developer misconception is that integer character constants have type **char** rather than **int**. Apart from the C++ compatibility issue and character constants appearing as the immediate operand of the **sizeof** operator, it is unlikely that there will be any cascading consequences following from this misconception.

---

884 The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer.

character constant value

Commentary

This mapping can occur in one of two contexts— translation phase 1, the results of which are used during preprocessing, and during translation phase 5. There is no requirement that the two mappings be the same.

116 translation phase 1
133 translation phase 5
1874 footnote 141

```
1   #include <stdio.h>
2
3   void f(void)
4   {
5   if ('a' == 97)
6      printf("'a' == 97 in translation phase 5\n");
7
8   #if 'a' == 97
9   printf("'a' == 97 in translation phase 1\n");
10  #endif
11  }
```

The mapping does not apply to any character constants that are denoted using octal or hexadecimal escape sequences (a fact confirmed by the response to DR #017, question 4 and 5).

Coding Guidelines

While translators treat character constants as numeric values internally, should developers be able to treat such constants as numeric values rather than symbolic quantities? The value of a character constant, which is not specified using an escape sequence, is representation information that depends on the implementation. (Although some of their properties are known, their range is specified and the digit characters have a contiguous encoding.) The guideline recommendation dealing with the use of representation information is applicable here.

882 escape sequence value within range
223 digit characters contiguous
569.1 representation information using

In some application domains a single representation is used for an implementations execution character set. Many character sets order the bit representations of their characters so that related characters have very similar bit patterns. For instance, corresponding upper- and lowercase letters are differentiated by a single bit in the Ascii character set, information that developers sometimes make use of to convert between upper- and lowercase (even though there are library functions available to perform the conversion). The following discussion looks at the issues involved in making use of character set representation details.

On seeing an arithmetic, bitwise, or relational operation involving a character constant, a reader needs to perform additional cognitive work that is not usually needed for other kinds of operations on this kind of operand, including:

- Performing a cognitive task switch. Character constants are usually thought of in symbolic rather than numeric terms. This is the rationale behind the lack of a guideline recommending that character constants be denoted, in the visible source by names, because of the strong semantic associations to readers of the source, created from experience in reading text.

0 cognitive switch
866 character constant syntax

- Recalling knowledge of the expected execution-time representation of character values. This is needed to deduce the intended consequences of the operation; for instance, using a bit-or operator to convert uppercase to lowercase.
- Deciding whether the result should continue to be treated as a symbolic quantity, or whether further operations are numeric in nature.

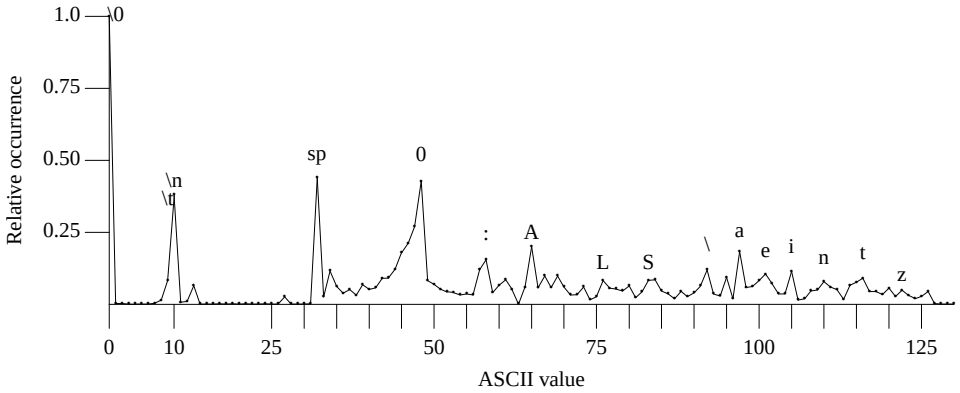


Figure 884.1: Relative frequency of occurrence of characters in an integer *character-constant* (as a fraction of the most common character, the null character). Based on the visible form of the `.c` files.

representation information using 569.1

An integer character constant appearing as the operand of a bitwise, arithmetic, or relational operator is making use of representation information and is covered by a guideline recommendation.

Example

```
#define mk_lower_letter(position) ('a' + (position))

void f(char p1, int p2)
{
if ('a' < 'b')
   ;
if ('a' < 97)
   ;

p1++;          /* Was p1 last assigned a decimal constant or an integer character constant? */
p1 = 'a' + 3;  /* Two different representation of an int being added. */
p1 = 'a' * 2;  /* What does multiplying 'a' by 2 mean? */
p1 = 'a' + p2; /* The value of p2 is probably not known at translation time. */
}
```

Table 884.1: Occurrence of a *character-constant* appearing as one of the operands of various kinds of binary operators (as a percentage of all such constants; includes escape sequences). Based on the visible form of the `.c` files. See Table 866.3 for more detailed information.

Operator	%
Arithmetic operators	4.5
Bit operators	0.5
Equality operators	31.3
Relational operators	4.1

character constant more than one character

The value of an integer character constant containing more than one character (e.g., **'ab'**), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. 885

Commentary

Why does the C Standard allow for the possibility of more than one character in an integer character constant? There is some historical practice in this area. There is also a long-standing practice (particularly in Fortran)

of packing several characters into an object having a larger integer type.

C90

The value of an integer character constant containing more than one character, or containing a character or escape sequence not represented in the basic execution character set, is implementation-defined.

C++

The C++ Standard does not include any statement covering escape sequences that are not represented in the execution character set. The other C requirements are covered by words (2.13.2p1) having the same meaning.

889 wide character escape sequence implementation-defined

Other Languages

A character constant, in the lexical sense, that can contain more than one character is unique to C.

Common Implementations

Given that an integer character constant has type **int**, most implementations support as many characters as there are bytes in that type, within character constants. The ordering of the characters within the representation of the type **int** varies between implementations. Possibilities, when the type **int** occupies four bytes, include (where underscores indicate padding bytes):

```
  'ab' ⇒ __ab, ab__, ba__
'abcd' ⇒ abcd, badc, dcba
```

The implementation-defined behavior of most implementations, for an escape sequence denoting a value outside of the range supported by the type **char** (but within the supported range of the type **unsigned char**), is to treat the most significant bit of the value as a sign bit. For instance, if the type **char** occupies eight bits, the value of the escape sequence `'\xFF'` is -1— if it is treated as a signed type. If it is treated as an unsigned type, the same escape sequence would have value 255. The sequence of casts `(int)(char)(unsigned char)0xFF` is one way of thinking about the conversion process.

Coding Guidelines

The representation, in storage during program execution, of an integer character constant containing more than one character depends on implementation-defined behavior; it also defeats the rationale of using a character constant (i.e., its character'ness). The guideline recommendation dealing with the use of representation information is applicable.

569.1 representation information using

Table 885.1: Number of *character-constant*s containing a given number of characters. Based on the visible form of the `.c` files.

Number of Characters	Occurrences	Number of Characters	Occurrences
0	27	4	21
1	50,590	5	4
2	0	6	4
3	8	7	0

886 If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.

character constant single character value

Commentary

char 477 hold any member of execution character set
basic character set 478 positive if stored in char object
escape sequence 882 value within range
char 516 range, value, implementation and behavior
escape sequence 882 value within range

A member of the basic execution character set can be represented in an object of type **char**, and all such members are represented using positive values. Under this model an escape sequence can be thought of as representing a sequence of bits. The value of this bit representation must be representable in the type **unsigned char**, but the actual value used is the bit representation treated as having a type **char**. The type **char** supports the same range of values as either the signed or unsigned character types, and the value of an escape sequence must be representable in the type **unsigned char**. If the type **char** is treated as being signed, any escape sequences whose value is greater than SCHAR_MAX results in implementation-defined behavior.

C++

The requirement contained in this sentence is not applicable to C++ because this language gives character literals the type **char**. There is no implied conversion to **int** in C++.

Other Languages

In most other languages character constants have type **char** and these issues do not apply.

Coding Guidelines

It is possible that a character constant may denote a value that is not representable in some implementations **char** type. An implementation's inability to support the desired behavior for these values is an issue that may need to be taken into account when selecting which vendors' translator to use. A guideline recommending against creating such a character constant serves no practical purpose (it is assumed that such a value was denoted using a character constant for a purpose).

---

wide character constant type of

A wide character constant has type **wchar_t**, an integer type defined in the **<stddef.h>** header. 887

Commentary

It is the responsibility of the implementation to ensure that the definition of wchar_t contained in the **<stddef.h>** header is compatible with the internal type used by the translator to assign a type to wide character constants. A developer-defined typedef whose name is wchar_t does not have any affect on the type of a wide character constant, as per the behavior for **sizeof** and size_t.

sizeof 1127 result type

C++

2.13.2p2 *A wide-character literal has type **wchar_t**.[23)]*

In C++ **wchar_t** is one of the basic types (it is also a keyword). There is no need to define it in the **<stddef.h>** header.

3.9.1p5 *Type **wchar_t** shall have the same size, signedness, and alignment requirements (3.9) as one of the other integral types, called its underlying type.*

Although C++ includes the C library by reference, the **<stddef.h>** header, in a C++ implementation, cannot contain a definition of the type **wchar_t**, because **wchar_t** is a keyword in C++. It is thus possible to use the type **wchar_t** in C++ without including the **<stddef.h>** header.

Other Languages

In Java all character constants are capable of representing the full Unicode character set; there is no distinction between ordinary characters and wide characters.

Common Implementations

The type wchar_t usually has a character type on implementations that only support the basic execution character set.

Coding Guidelines

A program that contains wide characters is also likely to use the type `wchar_t` and the coding guideline issues are discussed under that type.

---

888 The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **`mbtowc`** function, with an implementation-defined current locale.

multibyte character mapped by mbtowc

Commentary

This requirement applies to members of the extended character set, not to members of the basic source character set.

216 extended character set

From the practical point of view, translators need at least some of the members of the basic source character set to always map to the same values. For instance, the format specifiers used in the `scanf` family of functions treat white space as a special directive. Multibyte characters in the format string are converted to widechars in parsing the format, but there is no `iswspace` library function to test for these characters. The only method is to check whether a character value is within the range of the basic execution character set and use the `isspace` library function. The translator behaves as-if the program fragment:

```
1  save_locale();
2  setlocale("LC_ALL", current_locale);
3  value = mbtowc(wide_character_constant);
4  restore_locale();
```

were executed in translation phase 7, where `curr_locale` is the implementation-defined current locale. The call to the `mbtowc` function cannot use the C locale by default because that locale does not define any extended characters. A related issue is discussed elsewhere.

215 extended characters
2024 L'x' == 'x'

C++

The value of a wide-character literal containing a single `c-char` *has value equal to the numerical value of the encoding of the* `c-char` *in the execution wide-character set.* 2.13.2p2

The C++ Standard includes the `mbtowc` function by including the C90 library by reference. However, it does not contain any requirement on the values of wide character literals corresponding to the definitions given for the `mbtowc` function (and its associated locale).

There is no requirement for C++ implementations to use a wide character mapping corresponding to that used by the `mbtowc` library function. However, it is likely that implementations of the two languages, in a given environment, will share the same library.

Coding Guidelines

Accepting the implementation-defined behavior inherent in using wide string literals is part of the decision process that needs to be gone through when using any extended character set. One of the behaviors that developers need to check is whether the implementation-defined locale used by the translator is the same as the locale used by the program during execution.

---

889 The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

wide character escape sequence implementation-defined

Commentary

Support for wide character constants containing more than one multibyte character is consistent with such support for integer character constants. This specification requires the multibyte character to be a member of the extended execution character set. The equivalent wording for integer character constants requires that the value map to single-byte execution character.

885 character constant more than one character

C++

The C++ Standard (2.13.2p2) does not include any statement covering escape sequences that are not represented in the execution character set.

Example

```
#include <wchar.h>

wchar_t w_1 = L'\xff',
        w_2 = L'ab';
```

EXAMPLE 1 The construction **'\0'** is commonly used to represent the null character. 890

Commentary

null character 219 There are many ways of representing the null character. The construction '\0' is the shortest *character-constant*. This character sequence is also used within string literals to denote the null character.

EXAMPLE 2 Consider implementations that use two's-complement representation for integers and eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant **'\xFF'** has the value **-1**; if type **char** has the same range of values as **unsigned char**, the character constant **'\xFF'** has the value **+255**. 891

Commentary

This difference of behavior, involving hexadecimal escape sequences, is often a novice developer's first encounter with the changeable representation of the type **char**.

EXAMPLE 3 Even if eight bits are used for objects that have type **char**, the construction **'\x123'** specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are **'\x12'** and **'3'**, the construction **'\0223'** may be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.) 892

Commentary

An example of what is specified in the syntax.

EXAMPLE 4 Even if 12 or more bits are used for objects that have type **wchar_t**, the construction **L'\1234'** specifies the implementation-defined value that results from the combination of the values **0123** and **'4'**. 893

wide character escape sequence implementation-defined 889

Commentary

What is meant by "combination of the values" is not defined by the standard. The construction L'\1234' might be treated as equivalent to the wide character constant L'S4' (*S* having the Ascii value 0123), or some alternative implementation-defined behavior may occur.

Forward references: common definitions **<stddef.h>** (7.17), the **mbtowc** function (7.20.7.2). 894

6.4.5 String literals

string literal syntax

895

string-literal:
" *s-char-sequence*$_{opt}$ "
L" *s-char-sequence*$_{opt}$ "

```
s-char-sequence:
            s-char

            s-char-sequence s-char
s-char:
            any member of the source character set except
                        the double-quote ", backslash \\, or new-line character
            escape-sequence
```

Commentary

Escape sequences are part of the syntax of string literals. This means that the consequences called out in footnote 64 also apply to string literals.

862 footnote 64

C++

In C++ a *universal-character-name* is not considered to be an escape sequence. It therefore appears on the right side of the *s-char* rule.

Other Languages

Some languages use the single-quote, ', character to delimit string literals. Character sequences were originally denoted in Fortran using the notation of a character count followed by the letter *H* (after Herman Hollerith, the inventor of the 80-column punch card), followed by the character sequence (e.g., `11HHello world`).

Common Implementations

When building a *string-literal* preprocessing token from a sequence of characters, most implementations simply search for the terminating *double-quote*. This means that escape sequences that are not part of the syntax, such as `\z`, are included as part of the string literal. Any multibyte characters contained within string literals and character constants are likely to be translated without diagnostic being issued. The execution-time behavior is dependent on the support provided by an implementation's library functions.

Most implementations issue a diagnostic if a *new-line* character is encountered before the terminating *double-quote* is seen. Syntax error recovery after this point can be very poor. Some implementations continue to look for a closing *double-quote*, while others terminate the string literal at that point. However, by then it is likely that characters intended to form other tokens have been subsumed into the string literal.

Coding Guidelines

Other coding guideline subsections discuss recommendations that symbolic names be associated with constant values. Do the same cost/benefit considerations also apply to string literals? The following list discusses the possibility of the benefits obtained by using symbolic names for other kinds of constants also providing benefits when applied to string literals:

825.3 integer constant not in visible source
842.1 floating constant not in visible source
866 character constant syntax
822 constant syntax

- *The string literal value may be changed during program maintenance.* In practice string literals are usually unique within the program that contains them. While it is unlikely that the same changes

will have to be made to identical string literals, relationships between other constants may exist. For instance:

```
#define MAX_NAME 12
#define NAME_FMT "%12s"   /* The digits must be the same as MAX_NAME. */
```

String literals are different from other constants in that it is sometimes necessary to make a change that relates to all of them. The common feature of string literals is usually the language in which their contents is written. (It may be decided to change messages to use the past tense rather than the present tense, add/remove full stops, or as an intermediate stage in localization.)

- *A symbolic name provides a more meaningful semantic association.* For some string literals it is possible to deduce a semantic association by reading the contained character sequence.
- *Reducing the cost of cognitive switches.* It is not possible to estimate whether reading the contents of the string literal, rather than reading a symbolic name to deduce a semantic association, incurs a greater or lesser cognitive switch cost.

While there may not be a worthwhile benefit in having a guideline recommending that names be used to denote all string literals in visible source code, there may be source configuration-management reasons why it may be worthwhile to place related string literals in a single file and reference them using a symbolic name. This consideration falls outside the scope of these coding guidelines.

Usage

Usage of escape sequences in string literal and string lengths is given elsewhere (see Table 866.2 and Figure 293.1).

Description

---

character string literal

A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in **"xyz"**. 896

Commentary

This is a restatement of information given in the Syntax clause which also defines the term *character string literal*.

The ordering of the sequence of characters in a source file forming a string literal token is honored in their ordering in storage. (There is no such guarantee for character constants containing more than one character.) The standard requires that implementations support a minimum number of characters in a string literal. limit 293 string literal

A string literal has uses other than for organizing characters to be sent to an output stream. The characters can also represent data that is used by algorithms within a program— for instance, the four letters ATCG representing the DNA bases in a program that recognizes sequences in genes.

Other Languages

Virtually every language has some form of string literal. Some languages enclose the characters in single-quotes.

Coding Guidelines

Any conversion by the translator of multibyte characters in string literals and character constants occurs in the locale of the translator. This locale need not be the same as the one that applies during program execution (which can be changed during program execution via calls to the `setlocale` library function). Ensuring consistency between translation- and execution-time locales is a software engineering issue that is outside the scope of these coding guidelines.

---

wide string literal

A *wide string literal* is the same, except prefixed by the letter **L**. 897

Commentary

This defines the term *wide string literal*. The prefix changes the type of the string literal. Like character string literals, the ordering of wide string characters in storage is the same as that given in the source code. The standard says nothing about the ordering of bytes within an object that has type `wchar_t`.

---

898 The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, except that the single-quote `'` is representable either by itself or by the escape sequence `\'`, but the double-quote `"` shall be represented by the escape sequence `\"`.

escape sequences string literal

Commentary

These issues are discussed in the subsection on character constants.

871 escape sequences character constant

Other Languages

The idea of doubling up on the string delimiter to represent a single one of these characters is used in some languages, while others use some form of escape sequence. Pascal represents a `'` character within a string literal using the notation ''. The null string is represented by '', and '''' represents a string containing one single-quote, not two null strings. Pascal also uses the `'` character to delimit both character constants and string literals.

Coding Guidelines

The same possibilities for confusing sequences of escape sequences applies to string literals as integer character constants.

871 escape sequences character constant

Semantics

---

899 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence.

Commentary

Being able to concatenate character and wide string literals is needed in a number of situations, including:

135 translation phase 6

- Macros in the header **<stdint.h>** define character string literals for use in formatted I/O; these may need to be appended to wide string literals that are also part of the format specifier.
- The preprocessor macros `__DATE__`, and `__TIME__`, may need to be appended to wide string literals.

C90

The C90 Standard does not allow character and wide string literals to be mixed in a concatenation:

> *In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string literal tokens, are concatenated into a single multibyte character sequence.*

The C90 Standard contains the additional sentence:

> *If a character string literal token is adjacent to a wide string literal token, the behavior is undefined.*

C90 does not support the concatenation of a character string literal with a wide string literal.

C++

2.13.4p3

In translation phase 6 (2.1), adjacent narrow string literals are concatenated and adjacent wide string literals are concatenated. If a narrow string literal token is adjacent to a wide string literal token, the behavior is undefined.

The C++ specification has the same meaning as that in the C90 Standard. If string literals and wide string literals are adjacent, the behavior is undefined. This is not to say that a translator will not concatenate them, only that such behavior is not guaranteed.

Other Languages

In some languages white space acts as a concatenation operator, while in some other languages the token `+` is used to indicate concatenation. Whichever method is used to indicate the operation, it usually takes place during program execution, although some implementations may carry out optimizations that perform it during translation.

Example

```
1  char *p1 = "A very long string ..."
2             "that needs to be split across a line";
3
4  char *p2 = "\0xff" "f"; /* Concatenation happens after processing of escape sequences. */
```

Usage

In the visible form of the `.c` files 4.9% (`.h` 15.6%) of all string literals are concatenated (i.e., immediately adjacent to another string literal) and 1.4% (`.h` 10.7%) occupied more than one source line (i.e., line splicing occurred).

line splicing 118

If any of the tokens are wide string literal tokens, the resulting multibyte character sequence is treated as a wide string literal; 900

Commentary

This requirement can be viewed as a promotion of the string to its widest version. There is no requirement that when a character string literal and wide string literal are concatenated, the elements in the character string literal be converted as if by a call to the `btowc` function. Also, there is no requirement that `ch == wctob(btowc(ch))` when `ch` is a member of the basic character set. The meaningfulness of the resulting wide string literal will depend on the mapping used for the basic character set (i.e., does `L'x' == 'x'` hold).

L'x' == 'x' 2024

Example

```
1  #include <stddef.h>
2
3  wchar_t *wp = L"My ascii friend said: " "Hello " L"Tamai San";
```

---

otherwise, it is treated as a character string literal. 901

Commentary

There is no reason to treat it otherwise.

Example

```
1  char *long_string = "Strings that span more than one"
2                      "line are best split up to make"
3                      "them more readable.";
```

902 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[66)]

string literal zero appended

Commentary

This zero value byte specifies how string literals are delimited in C. An alternative specification would have been to specify that a length byte, or word, should be placed before the first character of the string literal. This byte becomes part of the value of the string literal. It is always added, even if the multibyte character sequence already contains a byte with value zero anywhere within itself. However, there is a special case involving initializers where this byte is not added.

1700 EXAMPLE array initialization

Other Languages

Few languages specify the representation of string literals. Many implementations of Pascal originally used a byte at the front of the string to hold the number of characters in the literal. Using of a length byte limits the maximum length of a string literal. A number of Pascal implementations subsequently migrated to the null byte termination representation (which also provided compatibility with C).

Coding Guidelines

Experience has shown that developers regularly fail to take this zero value appearing at the end of a string literal into account. For instance, when allocating storage for a copy of a string literal, forgetting to add one to the value returned by the `strlen` function.

Example

```
1  char sa[4] = "abc";              /* sa[3] initialized with the value zero. */
2  char *two_strings = "abc" "def"; /* Zero value appended to the concatenated string literal. */
3  char *two_zeros = "1\000";       /* A zero byte is still appended. */
```

903 The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence.

string literal static storage duration

Commentary

151 static storage duration initialized before startup

This initialization occurs during program startup. The array of static storage duration is the sequence of storage locations used to hold the individual multibyte characters. Note that this storage area is not **const** qualified. Modifying a string literal is not prohibited, but it may have unexpected effects. Unlike the specification for compound literals, this array is not referred to as an unnamed object (although it is effectively one).

909 string literal modify undefined

1058 compound literal unnamed object

Other Languages

Most languages treat string literals as if they had static storage duration, but do not always explicitly call out this fact. Few languages discuss the underlying representation details of how string literals are allocated storage.

Common Implementations

Some freestanding implementations place string literals in read-only memory, often in the same storage area as the executable code. The reasons for this are economic as well as technical, read-only storage being cheaper than read/write storage. This usage also occurs on some hosted implementations, but is becoming rarer as vendors don't always want to give users read access to storage containing executable code (usually for security reasons).

A few hosted implementations mark the storage used for string literals as read-only. Such usage requires hardware support and only a few hosts provide such fine-grain memory management of program data.

string literal type

For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence; 904

Commentary

A character string literal is thus indistinguishable in storage from an array of **char** that has had each element assigned the appropriate character value, the last array element being assigned the value zero.

C++

2.13.4p1 *An ordinary string literal has type "array of n* `const char` *" and static storage duration (3.7), where n is the size of the string. . . .*

```
1  char *g_p = "abc"; /* const applies to the array, not the pointed-to type. */
2
3  void f(void)
4  {
5  "xyz"[1] = 'Y'; /* relies on undefined behavior, need not be diagnosed */
6  "xyz"[1] = 'Y'; // ill-formed, object not modifiable lvalue
7  }
```

Example

If the identifier `anonymous` denotes the arrays of static storage duration allocated by the implementation to hold string literals, then the following initializers are equivalent:

```
1  static char anonymous_a[] = "a\xFF";
2  static char anonymous_b[] = {(char)'a', (char)0xFF, 0};
```

wide string literal type of

for wide string literals, the array elements have type **wchar_t**, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the **mbstowcs** function with an implementation-defined current locale. 905

Commentary

The `mbstowcs` function maps from the representation used in the wide string to the wide character representation used for the type `wchar_t`. This function maps a sequence of such multibyte characters, while the `mbtowc` function specified for wide character constants operates on a single multibyte character. A developer-defined typedef whose name is `wchar_t` does not have any affect on the type of a wide character constant, as per the behavior for wide character constants.

multibyte character mapped by mbtowc 888

wide character constant type of 887

C90

The specification that `mbstowcs` be used as an implementation-defined current locale is new in C99.

C++

2.13.4p1 *An ordinary string literal has type "array of n* `const char` *" and static storage duration (3.7), where n is the size of the string. . . .*

The C++ Standard does not specify that `mbstowcs` be used to define how multibyte characters in a wide string literal be mapped:

2.13.4p5

The size of a wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating `L'\0'`.

The extent to which the C library function `mbstowcs` will agree with the definition given in the C++ Standard will depend on its implementation-defined behavior in the current locale.

Coding Guidelines

The issue of ensuring consistency of locales is discussed elsewhere.

888 multibyte character mapped by mbtowc

---

906 66) A character string literal need not be a string (see 7.1.1), because a null character may be embedded in it by a **\0** escape sequence.

footnote 66

Commentary

This footnote is highlighting an important distinction— a string is terminated by the first null character in a sequence of characters, while a character string literal is simply a sequence of characters enclosed between double-quote characters. A translator will still append a zero to a string literal, even if it already contains one.

902 string literal zero appended

C++

This observation is not made in the C++ document.

Other Languages

Many languages do not specify how strings are to be represented, but their implementations do need to select some method. Any implementation that chooses to indicate the end of a string by using a value that could be contained within a string literal will encounter this issue.

Coding Guidelines

The usual method of finding the end of a string literal is to search for the terminating null character. If a string literal contains more than one such byte, some other method of knowing the number of bytes may be needed. Such usage is certainly unusual and runs counter to common developer expectations (always a potential source of faults). However, there is little evidence to show that faults are introduced into programs because of the presence of multiple null characters in string literals.

Example

```
1  char *seven_nulls = "\0\0\0\0\0\0\0"; /* 7th added by translator. */
2  char *packed_alphabet = "a\0abilities\0ability\0able\0about";
```

---

907 The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

Commentary

A string literal is the sum of its parts. This specification of behavior mimics that given for character constants.

885 character constant more than one character
889 wide character escape sequence implementation-defined

C90

This specification of behavior is new in C99.

C++

Like C90, there is no such explicit specification in the C+ Standard.

Common Implementations

Most implementations map octal and hexadecimal escape sequences to their numeric value and copy this into the program image.

Usage

In the visible form of the .c files 2.1% (.h 2.9%) of characters in string literals are not in the basic execution character set (the value of escape sequences were compared using the values of the Ascii character set).

---

string literal distinct array

It is unspecified whether these arrays are distinct provided their elements have the appropriate values. 908

Commentary

string literal 1068 distinct object

A translator may assign string literals that contain the same values to the same storage locations. This has the advantage of reducing the total amount of storage required for static data. It is also possible to overlay string literals with characters at the end of other string literals. However, there is a possible interaction here with the **restrict** qualifier. Passing a string literal as an argument to a function whose parameter is a pointer to a restrict-qualified type could be undefined behavior, if the storage used to hold the string literal was shared by more than one reference to that literal.

restrict 1491 intended use

footnote 1073 82

string literal 1068 distinct object

There is a similar statement for string literals associated with compound literals.

C++

Clause 2.13.4p4 specifies that the behavior is implementation-defined.

Other Languages

Most languages do not permit the contents of string literals to be modified, so whether they share the same storage locations is not an issue.

Coding Guidelines

The developer does not usually have any control over how string literals are allocated in storage; but the developer can choose not to modify them.

string literal 909 modify undefined

Example

In the following code do p1 and p2 both initially point to the same static array? Does p3 == (p1 + 6)?

```
1  char *p1 = "Hello World";
2  char *p2 = "Hello World";
3  char *p3 = "World";
```

Table 908.1: Number of *string-literal*s (the empty *string-literal*, i.e., "", was not counted). Based on the visible form of the .c and .h files. Although many of the program source trees contain more than one program, they were treated as a single entity. A consequence of this is that the number of unique matches represents a lower bound; having a smaller number of string literals is likely to reduce the probability of matches occurring.

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql	Total
Number of strings	38,063	21,811	177,224	30,358	30,574	11,285	16,387	325,702
Bytes in strings	656,366	324,667	4,050,258	512,766	737,015	288,018	298,888	6,867,978
Number of unique strings	18,602	9,148	114,170	17,192	18,483	7,401	7,930	187,549
Bytes in unique strings	434,028	170,170	3,189,466	378,917	562,555	240,811	219,690	5,159,385

---

string literal modify undefined

If the program attempts to modify such an array, the behavior is undefined. 909

Commentary

If a string literal occupies storage that has been used to represent more than one string literal, then a modification of one string literal could affect the values of other string literals. If the string literal has been placed in read-only memory, any attempted modification will have no effect.

Other Languages

Most languages treat string literals as read-only data, although some (e.g., Fortran) do not specify that their values cannot be changed.

Common Implementations

Most hosted implementations allow the modification to take place without complaint. The decision on shared storage will have been made by the translator, and the host O/S or the implementation's runtime system is unlikely to have any say in the matter.

Coding Guidelines

The following deviation assumes the modification will have the desired effect.

0 deviations
coding guidelines

Cg 909.1

A program shall not modify a string literal during its execution.

Dev 909.1

If string literals occupy a substantial amount of the storage available to a program and worthwhile savings are obtained by allowing them to be modified, they may be modified.

Example

```
1  char *p1 = "Hello World";
2  char *p2 = "Hello World";
3  char *p3 = "World";
4
5  void f(void)
6  {
7  "Hello World"[3] = 'a';
8
9  *p1 = 'B'; /* Does the character pointed to by p2 now equal 'B'? */
10 *p3 = 'w'; /* Does p2 now point at the string Hello World? */
11 }
```

---

910 EXAMPLE This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are **'\x12'** and **'3'**, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

Coding Guidelines

This example shows the advantage of using an octal escape sequence in this context. A leading zero could have been given to ensure that the character '3' would not have been interpreted as belonging to that octal escape sequence.

---

911 **Forward references:** common definitions **<stddef.h>** (7.17), the **mbstowcs** function (7.20.8.1).

6.4.6 Punctuators

---

punctuator
syntax

912

punctuator: one of

```
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:%:
```

Commentary

In most cases it is only necessary for a lexer to lookahead one character when dividing up the input stream into preprocessing tokens (the need to potentially tokenize the punctuators **%:%:** and **...** requires two characters of lookahead, for when the last character is not one needed to build them).

The only graphic characters from the Ascii character set not included in this list are *$*, *@*, and \ (which is used to indicate an escape sequence).

C90

Support for **<: :> <% %> %: %:%:** was added in Amendment 1 to the C90 Standard. In the C90 Standard there were separate nonterminals for punctuators and operators. The C99 Standard no longer contains a syntactic category for operators. The two nonterminals are merged in C99, except for **sizeof**, which was listed as an operator in C90.

C++

The C++ nonterminal *preprocessing-op-or-punc* (2.12p1) also includes:

```
::   .*   ->*   new   delete
and   and_eq   bitand   bitor   compl
not   not_eq   or   or_eq   xor   xor_eq
```

The identifiers listed above are defined as macros in the header **<iso646.h>** in C. This header must be included before these identifiers are treated as having their C++ meaning.

Other Languages

C contains a much larger set of punctuators than most other programming languages. Some languages use the *$* character as an operator/punctuator (e.g., Snobol 4). One of the original design aims of Cobol was to make programs written in it read like English prose. To this end keywords were used to represent operators that were normally indicated by punctuators in other languages:

```
01 data-value PIC 9(3) OCCURS 10 TIMES.
MOVE x to b.
ADD x to y GIVING z.
WRITE report-out AFTER ADVANCING PAGE.
```

Some languages (e.g., Ada and Fortran) include the ** operator (usually as a binary operator denoting exponentiation). Fortran uses abbreviated names for some operators (e.g., **.EQ.** for **==** and **.LT.** for **<**).

Common Implementations

Some implementations include the *@* character in the list of punctuators.

Coding Guidelines

Some punctuators are visually very similar to other punctuators (e.g., **!=** and **|=**). Other punctuators are very easily overlooked— for instance, the comma character. One of the reasons for these visual similarities is the font used to display source. The importance of presenting characters in an easy-to-read form has been recognized in other disciplines; for instance, chemistry and mathematics have specific fonts that have been designed for them. While a great deal of effort has been invested in creating fonts that make it easier to read text documents, such as books, or grab people's attention (e.g., advertising) very little effort has gone into designing a font to make source code easier to read.

Baecker and Marcus[77] make a number of interesting suggestions. For instance, unary operators can be made more noticeable by having them in superscript (e.g., $i = j * {}^{-}k$ versus $i = j *^{-} k$ or $x = y + {}^{+}- z$ versus $x = y^{++} - z$). While some fonts do deal with character pairs, the pairs that occur in source tend to be separated by other characters; for instance, matching bracketing characters, such as **[ ]**, **( )**, and **{ }**, can be nested within each other. Having the outer brackets displayed in a larger point size (e.g., $\big(\big((\ldots)\big)\big)$) makes the job of matching them up much easier to do visually.

It would be impractical for these coding guidelines to recommend the use of particular fonts when any that have been tuned to the display of C source are not commonly available, or where such usage requires particular tool support.

Table 912.1: Commonly used terms for punctuators and operators.

Punctuator/ Operator	Term	Punctuator/ Operator	Term
`[ ]`	left square bracket or opening square bracket or bracket	`^`	circumflex or xor or exclusive *or*
`( )`	left round bracket or opening round bracket or bracket or parenthesis	`|`	vertical bar or bitwise *or* or *or*
`{ }`	left curly bracket or opening curly bracket or bracket or brace	`&&`	and and or logical and
`.`	dot or period or full stop or dot selection	`||`	logical *or* or *or*
`->`	indirect or indirect selection	`?`	question mark
`*`	times or star or dereference or asterisk	`:`	colon
`+`	plus	`;`	semicolon
`-`	minus or subtract	`...`	dot dot dot or ellipsis
`~`	tilde or bitwise not	`=`	equal or assign
`!`	exclamation or shriek	`*=`	times equal
`++`	plus plus	`/=`	divide equal
`--`	minus minus	`%=`	percent equal or remainder equal
`&`	and or address of or ampersand or bitwise-and	`+=`	plus equal
`/`	slash or divide or solidus	`-=`	minus equal
`%`	remainder or percent	`<<=`	left-shift equal
`<<`	left-shift	`>>=`	right-shift equal
`>>`	right-shift	`&=`	and equal
`<`	less than	`^=`	xor equal or exclusive or equal
`>`	greater than	`|=`	or equal
`<=`	less than or equal	`,`	comma
`>=`	greater than or equal	`#`	hash or sharp or pound
`==`	equal	`##`	hash hash or sharp sharp or pound pound
`!=`	not equal	`<: :>` `<% %> %:` `%:%:`	no commonly used terms

Table 912.2: Occurrence of *`punctuator`* tokens (as a percentage of all tokens; multiply by 1.88 to express occurrence as a percentage of all punctuator tokens). Based on the visible form of the `.c` and `.h` files.

Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens
`,`	8.82	`==`	0.53	`||`	0.16	`-=`	0.03
`)`	8.09	`:`	0.46	`+=`	0.11	`++v`	0.02
`(`	8.09	`-v`	0.40	`>`	0.11	`%`	0.02
`;`	7.80	`*p`	0.40	`<<`	0.09	`--v`	0.01
`=`	3.08	`+`	0.38	`?:`	0.08	`...`	0.01
`->`	3.00	`*v`	0.34	`?`	0.08	`>>=`	0.01
`}`	1.87	`&`	0.32	`|=`	0.08	`^`	0.01
`{`	1.87	`!`	0.31	`>=`	0.07	`+v`	0.00
`.`	1.26	`v++`	0.27	`/`	0.06	`%=`	0.00
`*`	1.10	`&&`	0.26	`>>`	0.06	`##`	0.00
`#`	1.00	`!=`	0.26	`~`	0.05	`*=`	0.00
`]`	0.96	`<`	0.22	`v--`	0.04	`/=`	0.00
`[`	0.96	`-`	0.19	`&=`	0.04	`<<=`	0.00
`&v`	0.58	`|`	0.17	`<=`	0.04	`^=`	0.00

Semantics

A punctuator is a symbol that has independent syntactic and semantic significance. 913

Commentary

preprocessing token syntax 770
preprocessing token converted to token 137
postfix expression syntax 985

The syntactic form of a punctuator that is a *preprocessing-token* also has the syntactic form of a punctuator that can be converted (in translation phase 6) to a *token*. Some punctuators have syntactic significance only (e.g., **;**), while others can occur in several contexts— for instance, the pair **( )** can be used to bracket expressions or to denote the function call operator.

C90

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value.

The merging of this distinction between operators and punctuators, in C99, makes no practical difference.

C++

This observation is not made in the C++ Standard.

operator

Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). 914

Commentary

This defines the term *operator*. The meaning of some punctuators is dependent on the context in which they occur. For instance, the **:** token can occur after a case label in a bit-field declaration, or as the second operator in a ternary expression using the **?** operator. (The **:** character can also appear as one of the characters in a digraph.)

C90

In the C90 Standard operators were defined as a separate syntactic category, some of which shared the same spelling as some punctuators.

An operator specifies an operation to be performed (an evaluation) that yields a value, or yields a designator, or produces a side effect, or a combination thereof.

An *operand* is an entity on which an operator acts. 915

Commentary

This defines the term *operand*, which is common to most languages.

C++

The nearest C++ comes to defining *operand* is:

5p1 *An expression is a sequence of operators and operands that specifies a computation.*

916 In all aspects of the language, the six tokens[67] digraphs

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

```
[   ]   {   }   #   ##
```

except for their spelling.[68]

Commentary

The intent of introducing digraphs was to produce more readable alternatives to trigraphs. The character sequences chosen are shorter and matching pairs do contain some symmetry.

C90

These alternative spellings for some tokens were introduced in Amendment 1 to the C90 Standard. As such there is no change in behavior between C90 and C99.

Other Languages

Digraphs are unique to C (and C++).

Common Implementations

Even though digraphs were first introduced in 1993, vendors have been slow to add support for them in translators.

The Perkin-Elmer C compiler[1075] treated the following character sequences:

```
(|  |)  (<  >)  \!!  \!  \(  \)  \^
```

as being equivalent, respectively, to the tokens (the last four character pairs were also treated as escape sequences): 866 escape sequence syntax

```
[   ]   {   }   ||   |   {   }   ~
```

Coding Guidelines

Digraphs have several advantages over trigraphs. They are more readable and they are not substituted for inside string literals and character constants. Their only disadvantage is the continuing lack of support in a large number of translators. The characters used to denote digraphs were chosen for the very low probability of their occurring in existing programs outside of string literals. It is possible for a program to exhibit behavior that depends on whether digraphs are supported or not. But such usage is likely to be rare and not worth a guideline recommendation.

Example

```
1  #include <stdio.h>
2
3  #define prog_code(x) prog_string(%: x)
```

```
#define prog_string(y) "prog: " #y

void c_feather(void)
{
printf("%s\n", prog_code(10));
}
```

Without support for digraphs the output is:

```
prog: %: 10
```

With support for digraphs the output is:

```
prog: "10"
```

Usage

The visible form of the `.c` files contained zero digraphs.

Forward references: expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8). 917

6.4.7 Header names

header name syntax

918

```
header-name:
                < h-char-sequence >
                " q-char-sequence "
h-char-sequence:
                h-char
                h-char-sequence h-char
h-char:
                any member of the source character set except
                                the new-line character and >
q-char-sequence:
                q-char
                q-char-sequence q-char
q-char:
                any member of the source character set except
                                the new-line character and "
```

Commentary

Headers enclosed between the < and > delimiters are commonly called a *system header* or *implementation header*. Such headers are generally special in that they are usually supplied by the implementation, host OS, or third-party API vendor. The " delimited form is commonly called a *header*.

Other Languages

While many languages do not specify any kind of header name tokens, their implementations usually add support for some such functionality as an extension. Use of double-quotes is commonly seen.

Usage

source file inclusion 1896

Header name usage information is given elsewhere.

Semantics

919

The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2.

Commentary

The *header-name* preprocessing token can only occur as part of a **#include** preprocessing directive. The issues involved in the implementation-defined mapping are discussed elsewhere.

924 header name recognized within #include
1896 source file inclusion

---

920 If the characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, the behavior is undefined.

characters between < and >delimiters

Commentary

The character sequences // and /* denote the start a comment. The character \ denotes the start of an escape sequence. The characters ' and " delimit character constants and string literals, respectively.

Preprocessing tokens and comments are handled in translation phase 3. The standard specifies no ordering dependencies on these operations. A lexical analyzer which has no knowledge of the context in which sequences of characters occur, would return a sequence of preprocessing tokens that subsequent processing (e.g., in translation phase 4) would need to join together to form a *header-name* preprocessing token. The characters listed above could all cause behavior such that a translator would not be able to join the necessary preprocessing tokens together in translation phase 4.

124 translation phase 3
129 translation phase 4

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

Other Languages

Because support for some form of **#include** directive is usually provided as an extension by language implementations, the issue of certain sequences of characters having special meaning within a header name is part of an implementation's behavior, not the language specification.

Common Implementations

Most translators maintain sufficient context information that they are aware that a sequence of characters occurs on the same line as a **#include** preprocessing directive. In this case translators have sufficient information to know that a *header-name* preprocessing token is intended and can act accordingly.

Coding Guidelines

The mapping issues involved with these characters are discussed elsewhere.

116 translation phase 1
1897 #include h-char-sequence

---

921 Similarly, if the characters ', \, //, or /* occur in the sequence between the " delimiters, the behavior is undefined.[69)]

Commentary

The issues involved are the same as those discussed in the previous sentence.

920 characters between < and >delimiters

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

Coding Guidelines

One difference between the " delimiter and the < and > delimiters is that in the former case developers are likely to have some control over the characters that occur in the *q-char-sequence*.

920 characters between < and >delimiters

---

922 67) These tokens are sometimes called "digraphs".

footnote 67 digraphs

Commentary

There is no other term in common usage; in fact many developers are not aware of the existence of digraphs.

---

923 68) Thus [and <: behave differently when "stringized" (see 6.10.3.2), but can otherwise be freely interchanged.

footnote 68

Commentary

operator 1958 All six digraphs and their respective equivalent tokens behave differently when stringized. Token glueing is not an issue because there is only one situation where it might be possible to glue two digraphs together to form another meaningful token, and the Committee has already specified that this case does not create a valid token. The term *stringize* is not defined by the standard. However, its common usage is as the name of the # operator.

EXAMPLE 1966
#

operator 1950

Coding Guidelines

Stringizing is a relatively rare operation and use of digraphs is even rarer. A guideline recommendation covering this case does not appear to be worthwhile.

Example

```
#define MK_STR(x) #x

char *p1 = MK_STR([);    /* Assigns the string "[" */
char *p2 = MK_STR(\??(); /* Assigns the string "[" */
char *p3 = MK_STR(<#);   /* Assigns the string "<#" */
```

header name recognized within #include

~~A header name preprocessing token is~~ Header name preprocessing tokens are recognized only within a **#include** preprocessing ~~directive.~~ directives or in implementation-defined locations within **#pragma** directives[DR324)]. 924

Commentary

header name 783 exception to rule

The consequences of this requirement are discussed elsewhere.

The wording was changed by the response to DR #324.

C90

This statement summarizes the response to DR #017q39 against the C90 Standard.

C++

The C++ Standard contains the same wording as the C90 Standard.

Common Implementations

All known translators implemented this requirement, even though it was not what C90 originally, technically specified. Most implementations simply take all characters on the line to the right of a **#include** and do their own special processing on it. In other contexts characters are processed through the usual preprocessing token creation machinery.

EXAMPLE tokenization

EXAMPLE The following sequence of characters: 925

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a *{* on the left and a *}* on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

Commentary

The tokenization formed for the character sequence `member@$` may be different when using implementations that support additional characters within an identifier preprocessing token.

926 **Forward references:** source file inclusion (6.10.2).

6.4.8 Preprocessing numbers

pp-number syntax

927

pp-number:
digit
. *digit*
pp-number digit
pp-number identifier-nondigit
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

Commentary

This syntax supports the creation of preprocessing tokens that have no meaning as integer or floating-point tokens. The rationale for this relaxed syntax was to simplify the lexing of characters into preprocessing tokens rather than to support the stringizing of rather unusual sequences of characters. This flexibility also allows constructs such as:

```
1  #define glue(a, b) a ## b
2
3  float f = glue(1e+, 20);
```

to work as expected. The *identifier-nondigit* is needed to support hexadecimal constants.

There have been a number of requests to WG14 to tighten up the syntax of *pp-number*. The stated aim being to reduce the number of cases where a sequence of characters are treated as a *pp-number*, but cannot be converted to the token constant. The Committee's response to these requests is given in DR #003.

C90

The C90 Standard used the syntax nonterminal *nondigit* rather than *identifier-nondigit*.

Rationale

C99 replaces *nondigit* with *identifier-nondigit* in the grammar to allow the token pasting operator, ##, to work as expected. Given the code

```
#define  mkident(s)  s ## 1m
/* ... */
int mkident(int) = 0;
```

if an identifier is passed to the `mkident` macro, then 1m is parsed as a single pp-number, a valid single identifier is produced by the ## operator, and nothing harmful happens. But consider a similar construction that might appear using Greek script:

```
#define  μμμμk(p)  p ## 1μ
/* ... */
int μk(int) = 0;
```

For this code to work, 1μ must be parsed as only one pp-token. Restricting pp-numbers to only the basic letters would break this.

Support for additional digits via UCNs is new in C99. Also support for *p* and *P* in a *pp-number* is new in C99.

C++

Support for *p* and *P* in a *pp-number* is new in C99 and is not specified in the C++ Standard.

Other Languages

Most languages restrict the sequence of characters that can occur in an integer or floating-point token to those that are meaningful numbers. But then most languages do not include a preprocessor. The *p* and *P* form of exponents is unique to C.

Example

The character sequences:

```
1   22 .7  3.1E-41 1.2.3  4E+5SOME_OBJ6e+7
```

form the *pp-numbers*:

```
    {22} {.7}  {3.1E-41} {1.2.3}  {4E+5SOME_OBJ6e+7}
```

not:

```
    {22} {.7}  {3.1E-41} {1.2} {.3}  {4E+5} {SOME_OBJ} {6e+7}
```

Description

A preprocessing number begins with a digit optionally preceded by a period (**.**) and may be followed by valid identifier characters and the character sequences **e+**, **e-**, **E+**, **E-**, **p+**, **p-**, **P+**, or **P-**. 928

Commentary

This is a restatement of information given in the Syntax clause.

C90

Support for the *P* form of exponent is new in C99.

C++

The C++ Standard does not make this observation and like C90 does not support the *P* form of the exponent.

Preprocessing number tokens lexically include all floating and integer constant tokens. 929

Commentary

This describes in words what is specified in the syntax.

C++

This observation is not made in the C++ Standard.

Common Implementations

They also lexically include the binary constants supported by some implementations (e.g., 0b01010101).

Semantics

A preprocessing number does not have type or a value; 930

Commentary

A preprocessing number is nothing more than a sequence of characters.

Other Languages

Languages that do not include a preprocessor usually give a value and a type to such tokens as soon as they are created, that is immediately after lexing.

---

931 it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

Commentary

A preprocessing number may acquire a type and a value prior to translation phase 7 if it occurs within a **#if** preprocessor directive.

136 translation phase 7
1880 #if operand type uintmax_*

Coding Guidelines

A *pp-number* that occurs within a **#if** preprocessor directive is likely to have a different type than the one it would be given as part of translation phase 7. The implications of this difference are discussed elsewhere.

1880 #if operand type uintmax_*

---

932 69) Thus, sequences of characters that resemble escape sequences cause undefined behavior.

footnote 69

Commentary

Escape sequences start with the character \, one of the characters whose appearance in a *header-name* causes undefined behavior. Developers may make an association between a sequence of characters starting with \ and escape sequences (which are not converted until translation phase 5, but at which time any *header-name* preprocessing token will have ceased to exist), which technically does not exist at the same time that the *header-name* preprocessing directive exists.

866 escape sequence syntax
133 translation phase 5

Common Implementations

On some translator host environments, in particular the MS-DOS file system, the \ character is the directory separator. Because of the large volume of existing source code using the MS-DOS directory naming conventions, many implementations also support it as a directory separator (treating it as equivalent to the separator actually used— e.g., / under Linux).

Example

```
1  #include "dir\phile.h"
```

---

933 DR324) For an example of a header name preprocessing token used in a **#pragma** directive, see Subclause 6.10.9.

footnote DR324

Commentary

This footnote was added by the response to DR #324.

2030 _Pragma operator

6.4.9 Comments

Commentary

Comments do not affect the behavior of a program. They are intended to be of use to developers or tools that read the source code containing them. A number of tools use comments to help them do their job. These usually work by specifying sequences of characters at the start of the comment, which act as flags that can be detected when the tool reads the source. Uses include the following:

- *Version control.* Some revision control tools, such as `rcs`, search for identification keywords while checking out source. Matches against such keywords (e.g., `$Revision$` or `$Date$`) cause the keyword to be replaced with the appropriate values.[135] A source file may contain such identification keywords in comments, or sometimes in string literals (e.g., `static const char rcsid[] = "$Header$";`). The advantage of string literals is that they appear in the translated object code.

- *Documentation extraction.* Special flag characters indicate how the remaining source text is to be treated (e.g., man page, TEX source, etc.[523]).

Other Languages

In some implementations of Basic, mostly interpreter-based and seen in early hobbiest computers, the program source code was held in storage and interpreted directly. Comments needed to be skipped during program execution. In such environments the presence of comments could adversely affect a programs performance.

Common Implementations

There have been a few implementations that allowed translator directives to be placed within comments. The majority of modern translators use the **#pragma** preprocessing directive for this purpose.

#pragma 1994 directive

Coding Guidelines

Many coding guideline documents[912, 1305] specify what they consider to be good commenting practices. Although the use of comments is generally considered to be a *good thing* to do, there have been no studies quantifying their costs or benefits. These coding guidelines do not get involved in making recommendations on how to comment. There are a number of reasons for this:

- Writing effective comments is a skill (these coding guidelines do not aim to teach skills).
- Automatic enforcement of any guideline recommendation dealing with comments is likely to be difficult. If some form of commenting were shown to have a worthwhile cost/benefit ratio, the corresponding recommendation would need to be handled through code reviews. (The state of the art in static analysis of comments is many years away from having the natural language semantics capability needed for automatic enforcement.)

The grammar of comment layout

Technically comments can appear between any two preprocessing tokens in the source. In practice comments invariably appear in only a small number of locations in the source, relative to other preprocessing tokens. Experience shows that most developers visually organize comments; there might be said to be a comment layout grammar. Whether the use of a comment grammar simply represents the original author's desire for a pleasing layout, is simply a by-product of following simple rules while writing code, or is a cost-effective mechanism for reducing the effort needed by subsequent readers to comprehend the source is not known.

The presence of comments can also affect the layout of the constructs to which they refer. Components of this comment grammar include:

- Comments *attached* to the source statement they refer to (e.g., visually located immediately above the source line it refers to, or to the right of it on the same line).
- Vertical alignment of comment boundaries. Visual neatness as an aid in associating one comment spread over multiple lines.
- *header* comments (e.g., at the start of a source file, or function definition) provide information about the sequence of lines/statements that follow it).

```
1  if (valu == 0) /* Comment to the right of the if it refers to. */
2     {
3     blah blah ; /* Comment about one statement. */
4
5  if (valu == 0) /* Comment about the if.               */
6     {           /* Continuing commentary about the if. */
7     blah blah ; /* and even more continuation of general commentary. */
```

```
8
9   /* Comment refers to statement below it. */
10  x=3;
11
12  /* Previous blank line helps delimit this comment from about C statement. */
13  y=3;
14
15  /*
16   * We could write a long comment that describes the conditions
17   * under which x and y take on certain values.  Developers are
18   * likely to associate the x & y mentioned in this comment with the
19   * identifier names in the if statement below.
20   */
21  if ((x == 3) && /* This comment implicitly refers to x. */
22      (y == 4))   /* Expression layout has been integrated into comment structure. */
```

Overview

Writing comments is a cost that has no short-term benefit for the developer who incurs that cost. They are purely an investment for a possible future benefit, probably to a different developer than the one who wrote them. Comments can reduce, and sometimes increase, the cost of comprehending source code.

Comments can reduce the effort needed to comprehend source code by providing information in a form that requires less effort to comprehend than source code. Comments can also increase the probability that subsequent modifications are correct by providing background information (i.e., on intended affects) that is not explicitly contained in the adjacent source code.

Comments can increase the effort needed to comprehend source code by providing incorrect, or misleading, information. The presence of comments can also increase the effort needed to visually scan source code. The visual organization of comments often has a structure that is separate from the grammar of the surrounding statements and declarations.[1397]

Comments are used for a variety of purposes, including:

- *To contain management information.* This can include the change history of a file, or function, and the person responsible. It might also be cross-references to other source files and documents, the comments effectively providing a means of embedding links within a source file.
- *To present information in a diagramatic fashion* (the hope being that this alternative form of presentation will be easier for the reader to interpret than source code).
- *To present information in natural language prose.* For readers new to the source code, the use of natural language may provide a more direct route to a reader's model of the world than the code itself.
- *To create place markers in the source.* For instance, placing a comment on the closing brace of a compound block to indicate the kind of statement that occurs at the start of the block (if, while, switch, etc.). Another example is the specification of source yet to be written at that point.

Documentation in comments

The quantity of documentation associated with programs can vary enormously. The case of no documentation, outside of the source code, is not uncommon, while for some large projects a large percentage of the total effort went into creating documentation.[166] Having this kind of information within the same file as the source code has several advantages:

- If the source code is available, the documentation is available (developers will be familiar with programs whose documentation has been lost, or requires significant effort to obtain).
- The colocation of documentation and source code might be thought to ensure that both are updated together (experience suggests that this is not always the case).

cost/accuracy trade-off

- People perform a cost/accuracy calculation when deciding where to obtain information from. Having documentation available in the source file reduces information access cost (compared to having to obtain it from another file), potentially leading to increased accuracy of the information used.

comment disadvantages

It also has disadvantages:

- Information within comments, which duplicates what appears in other documents, runs the risk of not being updated when the original document is changed, or vice versa. Duplicating information creates the problem of keeping both up-to-date, and if they are differences between them, knowing which is correct.
- Opinions formed early, based on limited data, interfere with information presented later using more accurate data.

belief maintenance

The environment in which developers work can also have an impact. An environment that provides support services for ensuring that documentation is maintained and readily available may have less of a need to duplicate large amounts of information in comments. In an environment where support services are poor and developers are responsible for providing and maintaining their own documentation, there are plenty of reasons why this task may not get done. When support services are poor, having documentation within source code may be the only way of ensuring that this information is available to subsequent maintainers.

literate programming

Literate programming

An extreme form of colocating source code and documentation is espoused by the so-called *literate programming*[752] approach to documentation. Here the source code and its associated documentation are kept in a single file. Various tools are used to separate out the program source and its documentation. The development method proposed by Knuth has gained some influential supporters and needs to be discussed. The choice of the term *literate programming* expresses an admirable intent. However, by concentrating on the end result— a beautifully laid out, typeset program and associated documentation— Knuth has completely ignored the context where most of a developer's interaction with source code occurs (i.e., reading the original source). The original *literate programming* source from which both the source code and program documentation are extracted is much more difficult to read than either of the two documents it contains.

The target market for Knuth's form of literate programming is widespread publication of source and documentation, where it is expected to be read by many people, usually for educational or training purposes. Comprehension of the contents of the original material (in its single file) only need be performed by a small number of people, and is not required to be simple or easy to do. Much of the published praise of Knuth's literate programming has concentrated on the quality of the final output documents. There is no published research comparing the costs of maintaining two separate documents versus a single document from which both the code and documentation is extracted. Your author can see no obvious advantages to using Knuth's literal programming approach to software development in a commercial environment. In this environment there are a small readership of the code and documentation, and most of the time is spent working with this material directly.

Reducing source code comprehension costs

The effectiveness of a comment has to be measured by the extent to which it reduces the cost of developer comprehension of the surrounding source code. Writing effective comments is a different kind of skill than writing code. Developers receive feedback when they write code, at the very least the translator issues diagnostics if they violate a constraint or syntax rule. The only way developers receive feedback on the effectiveness of their comments is when other people read them. In the short-term, the only time when this is likely to occur is during code review. The following are some of the cost/benefit analysis issues that apply to the use of comments:

- *Practice and feedback are needed to learn to write effective comments.* Is the cost of learning to comment, including the cost of poor commenting that will have been written in source during that

period, significantly less than the benefits likely to be obtained later? The issue of who pays this cost also needs to be considered in light of the probability of trained developers changing jobs.

- *Comments may need to be updated when the source they refer to is modified.* Does the person making the source code change know enough to be able to make an associated, meaningful change to the comment? What is the cost associated with updating all of the related comments? As source code nears the end of its useful life, it may be more cost effective to delete comments when the source they refer to is modified rather than updating them. (The update cost is not likely to be recouped; deleting comments removes the potential liability caused by them being incorrect.)
- *The liabilities of incorrect, or out-of-date, comments.* If the contents of comments disagrees with the behavior of the source code developers become confused and need to spend extra time deducing which is correct. This deduction process may reach an incorrect conclusion, possibly leading to faults being introduced.
- *How much benefit do comments provide?* There has been no study, using experienced developers, that has attempted to measure the benefit of comments.

A study by Roediger, Jacoby, and McDermott[1176] looked at the false memories created by misleading information. They found that memories of past events are influenced by previous recollections of those events. Also information retrieved during the most recent account of an event may have a larger effect on the current recollection than the original event itself.

Self commenting code

There is a school of thought that claims it is possible to write self-documenting programs, rendering comment redundant. It is just a matter of choosing the right names for identifiers. The function

```
1  unsigned int square_root(unsigned int valu)
2  {
3  unsigned int root = 0;
4
5  valu = (valu + 1) >> 1;
6  while (root < valu)
7     valu -= root++;
8
9  return root;
10 }
```

contains no comments, but its name makes it obvious to readers what it does. Is that sufficient? Are we going to trust that this algorithm really does return the square root of its argument? A comment giving a brief outline of the mathematics behind the algorithm might increase confidence in its behavior. IEC 60559 requires that implementation support a square root operation. A comment could explain that for certain ranges of argument this function is faster than making use of hardware-supported square root (which requires conversions to/from floating point, plus the actual square root operation).

A study of maintenance programmers by IBM[429] found that understanding the original authors intent was the most difficult problem they had.

Comment layout

Experience shows that developers do not like writing comments. Making it more difficult than it is already perceived to be could result in less time being spent in creating comments. Some developers invest a lot of time in formatting their comments. Whether or not this cost leads to any measurable benefit is debatable. However, some of the layouts used can require additional workload (compared to alternative layouts) should they need to be modified. For instance, in the comment

```
1  /*********************************************************
2   * Allocate a local variable of the given size.          *
3   * The offset depends on which way the system stack grows.*
```

```
4    * (Although the offsets are always positive)            *
5    *                                                       *
6    *  If the stack is ascending, we need to                *
7    *     i. Align on the correct boundary                  *
8    *    ii. The offset of the variable is the current offset*
9    *   iii. Increment the offset to allow for this variable *
10   *                                                       *
11   *  If the stack is descending, we                       *
12   *     i. Increment the current offset for this variable  *
13   *    ii. Align on the correct boundary                  *
14   *   iii. The offset of the variable is the current offset*
15   *                                                       *
16   * If the stack is descending, the offsets from the frame *
17   * pointer (fp) are negative within the interpreter but   *
18   * stored as positive offsets in mcc.  Thus aligning      *
19   * upwards on mcc's offsets actually aligns downwards in  *
20   * memory (which is what we require).                     *
21   *********************************************************/
```

the bounding of the text by star characters may be visually appealing (leaving aside the issue of whether any comment is important enough to warrant this degree of visual impact). But maintaining this layout, when the comment is updated, requires additional developer effort. Reducing the expected cost of writing a comment may increase the likelihood that no updates will be made to the contents of existing comments.

```
1    /*********************************************************
2       Lines can be added to this comment without the need to worry
3       about overrunning the length of the box that contains them,
4       or having to use tab/space as padding to add a terminating *
5
6       Existing wording can be edited without having to spend time
7       repositioning all those * characters.
8
9       The contained text is still visibly delimited.
10   *********************************************************/
```

Comment layout is a complex process[1397] and even the simplest of worthwhile recommendations are likely to be very difficult to enforce automatically (interested readers can find suggestions in other coding guideline documents[912, 1305]). The following recommendation is intended to help ensure that existing comments are kept up-to-date.

Rev 933.1 Comments shall not be laid out in a fashion that requires significant additional effort for developers wanting to modify their contents.

Visual affects

The presence of comments in source can create visual patterns that distract developers' attention away from patterns that may present in the declarations and statements. In the following example, the aligned comments down the right have the effect of creating two vertical visual groupings. The prominence of the horizontal empty space, separating declarations from statements, is reduced (see Figure 770.2).

```
1    #define BASE_COST 23             /* Production tooling costs.          */
2
3    extern int widgets_in_product;
4
5    int sum_widget_costs(void)       /* Return how much this will cost.    */
6    {
7    int num_widgets;                 /* A local count of widgets.          */
8    int total_cost;                  /* Running total of the costs.        */
9                                     /* this is the value returned.        */
```

```
10  widgets_in_product--;            /* Subtract one to make it zero-based. */
11
12  total_cost=BASE_COST;            /* A startup cost cannot be avoided.   */
13  if (widgets_in_product != 0)     /* make sure we have something to do.  */
14     { /* ... */ }
15  }
```

By drawing attention to itself, this form of blocked commenting has taken attention away from the declarations and statements. Comments have the lowest attention priority and their presentation needs to reflect this fact. Straker[1305] discusses visual effect issues in more detail.

Example

```
1  extern int *ptr;
2
3  void f(void)
4  {
5  int loc = 4 /*ptr; /* Blah blah. */;
6  }
```

Usage

While over 30% of the characters in this book's benchmark programs (see Table 770.3) are contained within comments, they only represent around 2% of the tokens. A study by Fluri et al[432] of the releases of three large Java programs over a 6 year period (on average) found three different patterns in the ratio of number of comment lines to number of non-comment lines for each program.

A study of comments in C++ source by Etzkorn[399] found that 57% contained English sentences (that could be automatically parsed by the tool used).

Table 933.1: Common formats of nonsentence style comments. Adapted from Etzkorn, Bowen, and Davis.[399]

Style of Comment	Example
Item name— Definition	MaxLength— Maximum CFG Depth.
Definition	Maximum CFG Depth.
Unattached prepositional phrase	To support scrolling text.
Value definitions	0 = not selected, 1 = is selected.
Mathematical formulas	Can be Boolean expressions...

Table 933.2: Breakdown of comments containing parsable sentences. Adapted from Etzkorn, Bowen, and Davis.[399]

Percentage	Style of Sentence	Example
51	Operational description	This routine reads the data. Then it opens the file.
44	Definition	General Matrix— rectangular matrix class.
2	Description of definition	This defines a NIL value for a list.
3	Instructions to reader	See the header at the top of the file.

Comments are usually written in the present tense, with either indicative mood or imperative mood.

Table 933.3: Common formats of sentence-style comments. Adapted from Etzkorn, Bowen, and Davis.[399]

Part of Speech	Percentage	Example
Present Tense	75	
Indicative mood, active voice		This routine reads the data.
Indicative mood, active voice, missing subject		Reads the data.
Imperative mood, active voice		Read the data.
Indicative mood, passive voice		This is done by reading the data.
Indicative mood, passive voice, missing subject		Is done by reading the data.
Past Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine opened the file. or Opened the file.
Future Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine will open the file. or Will open the file.
Other	15	

comment /*

Except within a character constant, a string literal, or a comment, the characters /* introduce a comment. 934

Commentary

The exception cases are implied by the phases of translation.

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

Other Languages

All programming languages support some form of commenting. The character sequences used to introduce the start of a comment vary enormously; for instance, Basic uses the character sequence **rem**, while Scheme and many assembly languages use **;**. Java supports what is called a documentation comment: such a comment is introduced by the character sequence /**; the additional * character distinguishing it from a nondocumentation comment.

Coding Guidelines

This form of comment introduction enables the creation of multiline comments. This has the advantage of simplifying the job of writing a long comment and the disadvantage of comments sometimes terminating in unexpected locations in the source.

commenting out

Developers sometimes use the /* style of commenting to *comment out* sections of source to prevent it being translated and executed. This might be necessary, for instance, because the source code is not yet working properly, or because it has been decided that its functionality is not needed at the moment. Use of comments for this purpose is sometimes said to run the risk of having a nested comment change the intended effect. In practice, the change of behavior usually occurs during translation, not program execution; for instance, attempting to comment out the following sequence of statements

```
1  some_value = 3;
2  if (test_bit == 1)
3    total++; /* Special case. */
4  do_something();
```

will cause a syntax violation at the closing comment sequence placed after the call to `do_something`. The opening comment sequence placed before the assignment to `some_value` being terminated by the */ characters appearing in the comment within the source being commented out.

Although it might be thought that a syntax violation would be sufficient warning for the developer to look more closely at the source, experience suggests that developers can become confused to the extent of deleting the terminating comment characters without deleting the introductory comment characters (the syntax violation goes away). Using the /* form of comments to temporarily stop declarations or statements influencing the behavior of a program is a known root cause of faults.

Some implementations provide an option to support the nesting of comments. The standard explicitly states that comments do not nest. If it is necessary to prevent source code from being translated, either the **#if** preprocessing directive (because such directives nest), or the // form of commenting needs to be used. 939 footnote 70 936 comment //

Cg 934.1 The /* kind of comment shall not be used to comment out source code.

This guideline recommendation does not prevent the use of source code within comments where it is plainly intended to be part of the exposition of the comment— for instance:

```
/*
 * A discussion on some application issue and the following is an
 * example of one possible way of solving it:
 *
 * some_value = 3;
 * if (test_bit == 1)
 *   total++;
 * do_something();
 *
 * But we chose not to do it this way ...
 */
```

One way of distinguishing commented out code from code that is part of a comment is to examine the context and commenting style. A simple opening comment character sequence, followed by declarations or statements, followed by the closing comment character sequence is obviously commented out code. In the above case the presence of comment text that discusses the code might be viewed as sufficient to distinguish the usage (of course it could have been a comment that happened to precede the statements that was subsequently merged with the following comment). The use of star characters at the start of every line further reduces the probability that this is commented out code.

A practical way of measuring the probability of source code within a comment being commented out code is the ease with which it can be converted to executable machine code. Only having to remove a single line, at the start and end of the comment, gives a high probability of the source being commented out. In the preceding case it is necessary to delete the first four lines and last three lines of the comment, followed by deleting the star character at the start of every line. It is very unlikely to be commented out source.

When using the /* form of commenting, care has to be taken to ensure that the contents of the comment are easily distinguishable from what is outside the comment. The following example shows how poor layout might lead to confusion:

```
extern void farm(int *);

void f(void)
{
int loc;

/* We are now going to perform some calculation that involves
a complicated formula.  The details can be found in
barns (1999), which is the best reference */
farm(&loc);
}
```

There are a number of techniques developers can use to make comments and their contents appear as a distinct visual unit. Starting each line of comment with a character that rarely occurs at the beginning of a

noncomment line creates continuity. Making it easy for readers to match the opening and closing comment character sequences helps to create visual symmetry.

```
extern void farm(int *);

void f(void)
{
int loc;

/*
 * We are now going to perform some calculation that involves
 * a complicated formula.  The details can be found in
 * barns (1999), which is the best reference
 */
farm(&loc);
}
```

sentence-picture relationships

Comments sometimes contain a diagram. A theoretical discussion of the advantages of a diagram over a purely sentence-based description is given by Larkin and Simon.[811] Experimental verification that pictures can enhance text memory is provided by a study by Waddill and McDaniel.[659] Readers of the source often compare diagrams against sequences of statements in the source code. The intent of this comparison is to verify that the two representations are consistent with each other. The following discussion is based on studies by Clark and Chase,[243] and Carpenter and Just.[202]

In a study by Clark and Chase[243] subjects were shown a display consisting of a sentence and a picture. They had to quickly press a button indicating whether the sentence was true or false. The sentences were "star is above plus", "star is below plus", "star isn't above plus", and "star isn't below plus", and the same four sentences with the words *star* and *plus* swapped. The pictures had the form $\overset{*}{+}$, or $\overset{+}{*}$. These sentences and pictures can be combined into four different kinds of questions. The sentence could be true/false, they could also be affirmative (state that a relationship is true) or negative (state that a relationship is not true) (see Table 934.1).

Table 934.1: Four types of questions.

Statement Relative to Fact	Example
true-affirmative (TA)	star is above plus: $\overset{*}{+}$
false-affirmative (FA)	plus is above star: $\overset{+}{*}$
false-negative (FN)	star isn't above plus: $\overset{*}{+}$
true-negative (TN)	plus isn't above star: $\overset{*}{+}$

Clark and Chase created a model using four parameters (whether above/below was used, the sentence being true/false, the sentence being stated in a negative form was the sum of two parameters) to account for the differing delays in subjects' responses to the information displayed. The predicted response time for answering a question could be obtained by adding the delays required by the parameters.

Carpenter and Just built a model (known as *The Constituent Comparison Model*; each proposition within the mental representation is referred to as a constituent) that combined these four parameters into one. This model makes two assumptions:

1. The information content of sentences and pictures is assumed to be mentally represented in propositional form. A proposition can be affirmative or negative; for instance, "The star is above the plus" is represented as *(AFFIRMATIVE (ABOVE, STAR, PLUS))*, and "The star isn't above the plus" as *(NEGATIVE (ABOVE, STAR, PLUS))*. Propositions can be embedded within one another *e.g., {FORTUNATE [NEG (RED, DOTS)]}). Pictures are assumed to always be represented affirmatively; that is, the mental representation specifies what the picture is, not what it is not.

propositional form 1707

2. The comparison process, between the two propositional forms, uses a single mental operation—retrieve and compare. Corresponding constituents from the sentence and picture representations are retrieved and compared, pair by pair. A subject's delay in responding is determined by the number of these operations.

The algorithm for determining the number of retrieve and compare operations is:

A boolean flag is used to hold the result state of the comparison process; its initial value is assumed to be true. Every time a mismatch is encountered the flag changes state (it can flip-flop between them). The time required for the change-of-state operation is assumed to be small, compared to the retrieve-and-compare operation.

When the comparison of a constituent proposition mismatches, the following operations occur:

1. the flag changes state,
2. the mismatching constituent is tagged to indicate that when the restarted comparison process encounters it again, it should be treated as a match,
3. the comparison process goes back to the innermost constituents and starts comparing from where it first started.

The time taken to determine whether a sentence matches a picture is proportional to the number of comparison operations. Two consequences of the Just and Carpenter model are that the greater the number of mismatches, the greater the number of comparison operations needed, and those mismatches that occur later will require more comparison operations than those that occur earlier. These predictions are borne out by the response timing from a variety of studies.[243]

This model assumes that pictures are represented propositionally. Is this always the case? A study by MacLeod, Hunt, and Mathews[883] found that 23% of subjects maintained a visual representation of the picture and converted the sentences they read into a mental image of the picture described. Because these subjects used pictorial representations, the linguistic structure of the sentence (e.g., the use of a negative) could not affect their performance.

Example

```
1  /* A simple comment on a single line, so why was this style used? */
2
3  #define OBSCURE_TEN (2/*/*/*/*/*/5)
4
5  /*
6   * A comment with a simple, straight-forward, easy-to-understand
7   * format.  Hmmm, if *'s appear at the start of a line then why not // ?
8   */
```

Table 934.2: Occurrence of kinds of comments (as a percentage of all comments; last row as a percentage of all new-line characters). Based on the visible form of the `.c` and `.h` files.

Kind of Comment	`.c` files	`.h` files
/* comment */	91.0	90.1
// comment	9.0	9.9
/* on one line */	70.3	79.1
new-lines in /* comments	12.3	17.5

935 The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[70)]

comment contents only examined to

Commentary

translation phase 3 124
trigraph sequences phase 1 117
line splicing 118

Comments are processed in translation phase 3. Trigraphs and line splicing will already have been handled.

C++

The C++ Standard gives no explicit meaning to any sequences of characters within a comment. It does call out the fact that comments do not nest and that the character sequence // is treated like any other character sequence within such a comment.

2.7p1 *The characters /* start a comment, which terminates with the characters */.*

Other Languages

Some languages support the use of translator directives within comments. This directive might control, for instance, the generation of listing files, the alignment of storage, and the use of extensions. Java supports the use of HTML tags inside its documentation comments. A few languages (e.g., Common Lisp) support nested comments and the contents of this form of comment need to be examined to identify the start/end of each nested comment.

Common Implementations

The contents of comments are sometimes examined by tools that analyze source code like a translator. The SVR5 `lint` tool[1393] includes the following:

```
extern int select;

void f(void)
{
if (select == 3)
   {
   return;
   /* NOTREACHED */  // Indicate that we know the statement is not reached
   select--;
   }
switch (select)
   {
   case 3: select++;
   /* FALLTHRU */  // Indicate that we know the case falls through
   case 4: select++;
           break;
   }
}
```

Coding Guidelines

Skipping all characters until the sequence */ is found is a rather open-ended operation. Experience suggests that the /* */ style of comment is not ideally suited for single-line comments; for instance, developers sometimes omit the closing */ characters. Also the characters */ occupy two positions on a line, which sometimes causes lines to wrap on display devices. The // form of comments does not have these problems.

Example

```
/\
* line splicing occurs before the comment is recognized
*\
/

/* ??) is a trigraph.  More importantly so is *??/
/
```

936 Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character.

comment //

Commentary

Many comments occupy a single line. The only form of comment supported by C90 required an explicit end-of-comment delimiter. Experience showed that omission of this closing delimiter is a cause of program faults. Also source code editors often wrap long lines and the two characters needed to close a comment can generate the need to abbreviate a comment or to have to accept a wrapped line in the displayed source. This newly introduced form of commenting does not suffer from these problems.

The C preprocessor is often used as a general preprocessor for other languages. The introduction of // as a comment start sequence in C99 is likely to cause problems for some developers; for instance, Fortran format statements can contain sequences of these two characters.

C90

Support for this style of comment is new in C99.

There are a few cases where a program's behavior will be altered by support for this style of commenting:

```
1  x = a //* */ b
2        + c;
3
4  #define f(x) #x
5
6  f(a//) + g(
7  );
```

Occurrences of these constructs are likely to be rare.

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

Other Languages

This style of comment is supported in BCPL, C++, Java, and many languages created in the last 10 years. Ada and Basic support the same commenting concept, they terminate at the end of line; the character sequences being used are `--` and **REM**, respectively. Fortran has always supported this concept of commenting; source lines that contain the letter C in the fifth column are treated as comments.

Common Implementations

Many implementations supported this style of commenting in their C90 translators.

937 The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

Commentary

As previously pointed out, this statement is a simplification.

935 comment contents only examined to

C++

The C++ Standard includes some restrictions on the characters that can occur after the characters //, which are not in C90.

> *The characters // start a comment, which terminates with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required.*

2.7p1

A C source file using the // style of comments may use form-feed or vertical-tab characters within that comment. Such a source file may not be acceptable to a C++ implementation. Occurrences of these characters within a comment are likely to be unusual.

Coding Guidelines

The // comment form is expected to occupy a single physical source line. It often comes as a surprise to readers of the source if such a comment includes the line that follows it.

Cg 937.1 The physical line containing the // from of comment shall not end in a line splice character.

EXAMPLE 938

```
"a//b"                  // four-character string literal
#include "//e"          // undefined behavior
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();          // syntax error, not comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

Commentary

This example lists some of the more visually confusing character sequences that can occur in source code.

footnote 70) Thus, /* ... */ comments do not nest. 939

Commentary

translation phase 3 124 A comment is a single lexical entity, replaced by one space character in translation phase 3. Support for nested comments would require that they have an internal structure.

Other Languages

While few language specifications support the nesting of comments, some of their implementations do.

Common Implementations

Some pre-C Standard translators supported the nesting of comments and some translators[578] continue to provide an option to support this functionality (although support for such an option is not as common as for many other pre-C Standard behaviors).

Coding Guidelines

The /* comment form is sometimes used for *commenting out* source code. Such usage is error prone because comments might already occur within the source that is intended to be removed from subsequent translator processing (meaning that some of this source is not *commented out*). This issue is discussed elsewhere. Enabling a translator's support for nested comments is doing more than enabling an extension, it can also change the behavior of strictly conforming programs.

commenting out 934 extensions 95.1 cost/benefit

Example

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  if (/*/*/0*/**/1)
6     printf("On this implementation comments nest\n");
```

```
7   else
8     printf("On this implementation comments do not nest\n");
9   }
```

6.5 Expressions

940 An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. expressions

Commentary

This defines the term *expression*. The operators available for use in C expressions reflect the kinds of operations commonly supported by processor instruction sets. Processors often contain instructions for performing operations that have no equivalent in C. For instance, the Intel SSE extensions[627] to the Intel x86 instruction set support the SHUFPS instruction (see Figure 940.1). There is no equivalent operator in C.

Expressions in C differ from expressions encountered in most algebra books in that in C the representable range is finite and contains an additional representable quantity, NaN (a few advanced books[897] cover the issues of infinity and NaN). Some of the expression transformations that deliver the same result in mathematics can deliver different results in C. 339 NaN

Although the evaluation of an expression is often thought about by developers as if it was a single sequence of operations, the only sequencing requirements are those imposed by sequence points. 187 sequence points

Most expressions need to be evaluated during program execution. The generation of machine code to evaluate expressions is a very simple problem. The complexities seen in industrial-strength translators are caused by the desire to generate machine code that minimizes some attribute (usually either execution time, size of generated code, or electrical power consumed). It is known that the selection of an optimal sequence of instructions, for an expression, is an NP-complete problem[8] (even for a processor having a single register[172]). Algorithms (whose running time is linear in the size of the input) that minimize the number of executed instructions, or accesses to storage, are known for those expressions that do not contain common subexpressions (for processors without indirection,[1214] stack-based processors of finite depth,[173] and general register-based processors[7]). expression optimal evaluation 1712 common subexpression

Unless stated otherwise, the result type of an operator is either that of the promoted operand or the common type derived from the usual arithmetic conversions. 706 usual arithmetic conversions

C++

The C++ Standard (5p1) does not explicitly specify the possibility that an expression can designate an object or a function.

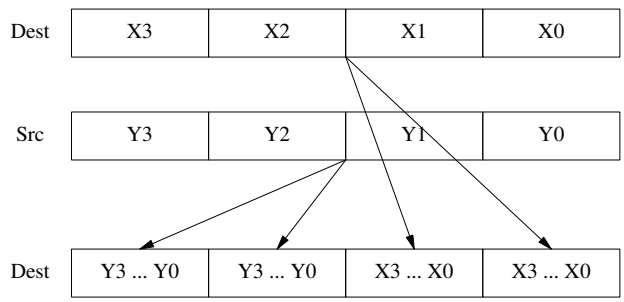


Figure 940.1: The SHUFPS (shuffle packed single-precision floating-point values) instruction, supported by the Intel Pentium processor,[627] places any two of the four packed floating-point values from the destination operand into the two low-order doublewords of the destination operand, and places any two of the four packed floating-point values from the source operand into the two high-order doublewords of the destination operand. By using the same register for the source and destination operands, the SHUFPS instruction can shuffle four single-precision floating-point values into any order.

Other Languages

This definition could be said to apply to most programming languages, which often contain a common core of similar operators operating on the same operand types. Languages vary in their support for additional operators and operand types that the core operators may operate on. A few languages (e.g., Algol 68, `gcc` supports compound statements in an expression) support the use of statements within an expression, or to be more exact, statements can return a value. Functional languages are side effect free. Evaluation of an expression returns a value and has no other effect.

Common Implementations

Evaluation of expressions is what causes translators to generate a large proportion of the machine code output to a program image. Optimization technology has improved over the years. The first optimizers operated at the level of a single expression, or statement. As researchers discovered new algorithms, and faster processors with more main memory became available, the emphasis moved to basic blocks. Here, the generation of machine code for expressions takes the context in which they occur into account. Optimizing register allocation, detecting and making use of common subexpressions, and on modern processors performing instruction scheduling to try to avoid pipeline stalls.[218] The continuing demand for more optimization has led to further research and commercialization of optimizers working at the so-called *super basic block* level, the level of a complete function definition, and most recently at the complete program level.

basic block 1710
register 1369 storage-class
common 1712 subexpression
basic block 1710
function 1821 definition syntax
translation unit 1810 syntax

Performance is often an issue in programs that operate on floating-point data. A number of transformations are sometimes used to produce expressions that deliver their results more quickly. These transformations can result in changes of behavior, including a greater range on the error bounds. Possible transformations are discussed under the respective operators. The Diab Data compiler[353] supports the `-Xieee754-pedantic` option to control whether these (i.e., convert divide to multiple) optimizations are attempted.

expression processor evaluation

Expression evaluation often requires that intermediate results be temporarily stored. The almost universal technique used is for processors to contain temporary storage locations— registers. (Stack architectures[762] may be simpler and cheaper to implement, but do not usually offer the cost/performance advantages of a register-based architecture, although they are occasionally seen in modern processors.[621, 1129]) The use of registers is discussed in more detail elsewhere.

register 1369 storage-class

The fact that operators are defined to operate on one or two values, returning a single value as a result, is not a hindrance to extending them to operate in parallel on vectors of operands as some extensions have done. In the quest for performance a number of processors are starting to offer vector operations. Operations to manipulate elements of a vector exist in these processors, but have yet to become sufficiently widely used to be explicitly discussed by the C Standard.

processor SIMD

There is no requirement in C that the operators in an expression be executed one at a time, although this is how the majority of processors have traditionally worked. The quest for performance has led processor vendors to try to perform more than one operation at the same time. Some vendors have chosen not to require translator support, the selection of the operations to execute in parallel being chosen dynamically by the processor (i.e., the latest members of the Intel x86 processor family[628]). Other vendors have introduced processors that operate on more than one data value at the same time, using the same instruction (known as *SIMD*, Single Instruction Multiple Data— pronounced sim-d). For such processors operations on arrays in a loop need not occur an element at a time; they can be performed 8, 16, 32, or however many parallel data operations are supported, elements at a time. Such processors require that translators recognize those situations where it is possible to perform the same operation on more than one value at the same time. SIMD processors used to be the preserve of up-market users, who could afford a Cray or one of the myriad of companies set up to sell bespoke systems. They are starting to become available as single-chip coprocessors for special embedded applications.[1448]

A description of how arithmetic operations are performed on integer and floating-point operands at the hardware level is given in Hennessy.[560]

Coding Guidelines

During almost all of human history, natural language has been used purely in a spoken form. The need to generate and comprehend sequences of words in realtime, using a less-than-perfect processor (the human brain), restricted the complexity of reliable communication. Realtime communication also provides an opportunity for feedback between speaker and listener, allowing content to be tailored for individual consumption.

It is often said that C source code is difficult to read. This is a special case of a more general observation. Most material created by people who are untrained in communicating ideas in written form is difficult to read. The written form of communication is not constrained by the need to be created in realtime, does not offer the opportunity for immediate feedback, and may have a readership that is not known to the author. An additional contribution to the *unreadability* of source code is that it is often written by people who consider themselves to be communicating with a computer rather than communicating with another person. Writers of C statements have the time needed to write complex constructs and the specifications they are given to work from are often complex. Duplicating this complexity often requires less effort than creating a simplified representation (i.e., copying can require less effort than creating).

A study by Miller and Isard[937] investigated subjects' ability to memorize sentences that varied in their degree of embedding. The following sentences are written with increasing amounts of embedding (the parse tree of two of them is shown in Figure 940.2).

1. She liked the man that visited the jeweler that made the ring that won the prize that was given at the fair.
2. The man that she liked visited the jeweler that made the ring that won the prize that was given at the fair.
3. The jeweler that the man that she liked visited made the ring that won the prize that was given at the fair.
4. The ring that the jeweler that the man that she liked visited made won the prize that was given at the fair.
5. The prize that the ring that the jeweler that the man that she liked visited made won was given at the fair.

The results showed that subjects' ability to correctly recall wording decreased as the amount of embedding increased, although their performance did improve with practice.

Other studies of reader's performance with processing natural languages have found the following:

- People have significant comprehension difficulties when the degree of embedding in a sentence exceeds two.[126]
- Readers' ability to comprehend syntactically complex sentences is correlated with their working memory capacity, as measured by the reading span test.[732] 1707 reading span
- Readers parse sentences left-to-right.[1083] An example of this characteristic is provided by so called *garden path* sentences, in which one or more words encountered at the end of a sentence changes the parse of words read earlier:

```
The horse raced past the barn fell.
The patient persuaded the doctor that he was having trouble with to leave.
While Ron was sewing the sock fell on the floor.
Joe put the candy in the jar into my mouth.
The old train their dogs.
```

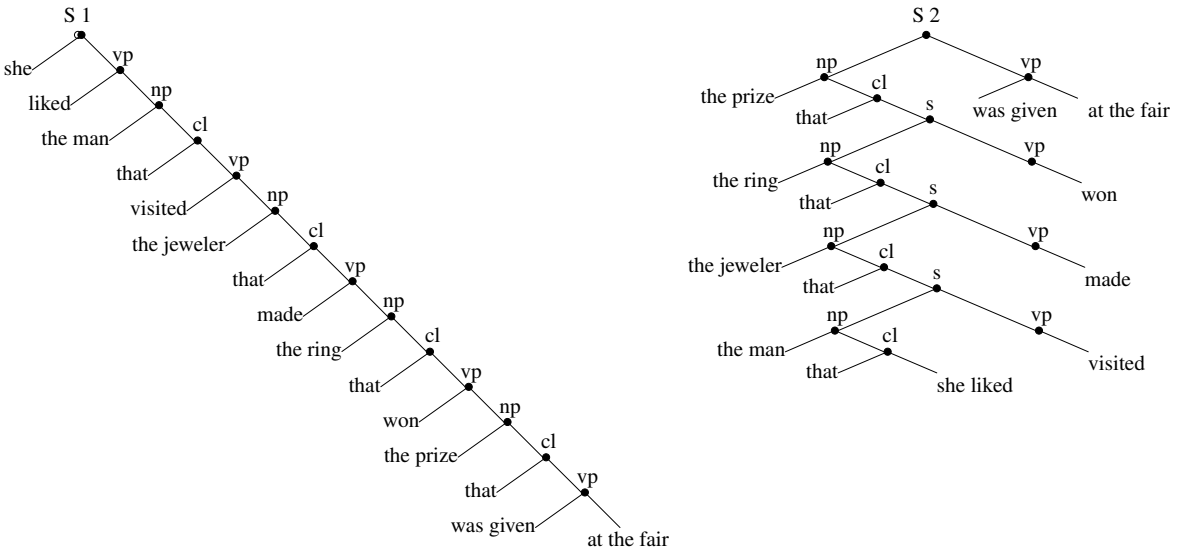


Figure 940.2: Parse tree of a sentence with no embedding (S 1) and a sentence with four degrees of embedding (S 2). Adapted from Miller and Isard.[937]

> In computer languages, the extent to which an identifier, operand, or subexpression encountered later in a full expression might change the tentative meaning assigned to what appears before it is not known.

How to readers represent expressions in memory? Two particular representations of interest here are the spoken and visible forms. Developers sometimes hold the sound of the spoken form of an expression in short-term memory; they also fix their eyes on the expression. The expression becomes the focus of attention. (This visible form of an expression, the number of characters it occupies on a line and possibly other lines, represents another form of information storage.)

Complicated expressions might be visually broken up into chunks that can be comprehended on an individually basis. The comprehension of these individual chunks then being combined to comprehend the complete expression (particularly for expressions having a boolean role). These chunks may be based on the visible form of the expression, the logic of the application domain, or likely reader cognitive limits. This issue is discussed in more detail elsewhere.

boolean role 476

memory 0
chunking

The possible impact of the duration of the spoken form of an identifier appearing in an expression on reader memory resources is discussed elsewhere.

identifier 792
primary
spelling issues
dead code 190
words 770
white space
between

Expressions that do not generate side effects are discussed elsewhere. The issue of spacing between tokens is discussed elsewhere. Many developers have a mental model of the relative performance of operators and sometimes use algebraic identities to rewrite an expression into a form that uses what they believe to be the faster operators. In some cases some identities learned in school do not always apply to C operators (e.g., if the operands have a floating-point type).

The majority of expressions contain a small number of operators and operands (see Figure 1731.1, Figure 1739.8, Figure 1763.1, and Figure 1763.2). The following discussion applies, in general, to the less common, longer (large number of characters in its visible representation), more complex expressions.

Readers of the source sometimes have problems comprehending complex expressions. The root cause of these problems may be incorrect knowledge of C or human cognitive limitations. The approach taken in these coding guideline subsections is to recommend, where possible, a usage that attempts to nullify the effects of incorrect developer knowledge. This relies on making use of information on common developer mistakes and misconceptions. Obviously a minimum amount of developer competence is required, but every effort is made to minimize this requirement. Documenting common developer misconceptions and then

recommending appropriate training to improve developers' knowledge in these areas is not considered to be a more productive approach. For instance, a guideline recommending that developers memorise the 13 different binary operator precedence levels does not protect against the reader who has not committed them to memory, while a guideline recommending the use of parenthesis does protect against subsequent readers who have incorrect knowledge of operator precedence levels.

943 operator precedence
943.1 expression shall be parenthesized

An expression might only be written once, but it is likely to be read many times. The developer who wrote the expression receives feedback on its behavior through program output, during testing, which is affected by its evaluation. There is an opportunity to revise the expression based on this feedback (assumptions may still be held about the expression— order of evaluation— because the translator used happens to meet them). There is very little feedback to developers when they read an expression in the source; incorrect assumptions are likely to be carried forward, undetected, in their attempts to comprehend a function or program.

The complexity of an expression required to calculate a particular value is dictated by the application, not the developer. However, the author of the source does have some control over how the individual operations are broken down and how the written form is presented visually.

Many of these issues are discussed under the respective operators in the following C sentences. The discussion here considers those issues that relate to an expression as a whole. While there are a number of different techniques that can be used to aid the comprehension of a long or semantically complex expression, your author does not have sufficient information to make any reliable cost-effective recommendations about which to apply in most cases. Possible techniques for reducing the cost of developer comprehension of an expression include:

- A comment that briefly explains the expression, removing the need for a reader to deduce this information by analyzing the expression.
- A complex expression might be split into smaller chunks, potentially reducing the maximum cognitive load needed to comprehend it (this might be achieved by splitting an assignment statement into several assignment statements, or information hiding using a macro or function).
- The operators and operands could be laid out in a way that visually highlights the structure of the semantics of what the expression calculates.

The last two suggestions will only apply if there are semantically meaningful subexpressions into which the full expression can be split.

Visual layout

expression visual layout

An expression containing many operands may need to be split over more than one line (the term *long* expression is often used, referring to the number of characters in its visible form). Are there any benefits in splitting an expression at any particular point, or in visually organizing the lines in any particular manner? There are a number of different circumstances under which an expression may need to be split over several lines, including:

- The line containing the expression may be indented by a large amount. In this case even short, simple expressions may need to be split over more than one line. The issue that needs to be addressed in this case is the large indentation; this is discussed elsewhere.
- The operands of the expression refer to identifiers that have many characters in their spelling. The issue that needs to be addressed in this case is the spelling of the identifiers; this is discussed elsewhere.
- The expression contains a large number of operators. The rest of this subsection discusses this issue.

1707 statement visual layout
792 visual skimming

Expressions do not usually exist in visual isolation and are not always read in isolation. Readers may only look at parts of an expression during the process of scanning the source, or they may carefully read an expression. (The issue of how developers read source is discussed elsewhere.) Some of the issues involved in the two common forms of code reading include the following:

770 reading kinds of

- During a careful reading of an expression reducing the cost of comprehending it, rather than differentiating it from the surrounding code, is the priority. Whether a reader has the semantic knowledge needed to comprehend how the components of an expression are mapped to the application domain is considered to be outside the scope of these coding guideline subsections. Organizing the components of an expression into a form that optimizes the cognitive resources that are likely to be available to a reader is within the scope of these coding guideline subsections. Experience suggests that the cognitive resource most likely to be exceeded during expression comprehension is working memory capacity. Organizing an expression so that the memory resources needed at any point during the comprehension of an expression do not exceed some maximum value (i.e., the capacity of a typical developer) may reduce comprehension costs (e.g., by not requiring the reader to concentrate on saving temporary information about the expression in longer-term memory). Studies have found that human memory performance is improved if information is split into meaningful *chunks*. Issues, such as how to split an expression into *chunks* and what constitutes a recognizable structure, are skills that developers learn and that are not yet amenable to automatic solution. The only measurable suggestion is based on the phonological loop component of working memory, which can hold approximately two seconds worth of sound. If the spoken form of a chunk takes longer than two seconds to say (by the person trying to comprehend it), it will not be able to fit completely within this form of memory. This provides an upper bound on one component of chunk size (the actual bound may be lower).

memory 0 chunking

phonolog-0 ical loop

- When scanning the code, being able to quickly look at its components, rather than comprehending it in detail, is the priority; that is, differentiating it from the surrounding code, or at least ensuring that different lines are not misinterpreted as being separate expressions. The edges of the code (the first non-white-space characters at the start and end of lines) are often used as reference points when scanning the source. For instance, readers quickly scanning down the left edge of source code might assume that the first identifier on a line is either modified in some way or is a function call.

Table 940.1: Occurrence of a token as the last token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). Based on the visible form of the `.c` files.

Token	% Occurrence of Token	% Last Token on Line	Token	% Occurrence of Token	% Last Token on Line
`;`	92.2	36.0	`#else`	89.1	0.2
* ... *\	97.9	8.4	`int`	5.3	0.2
`)`	20.6	8.3	`||`	23.7	0.2
`{`	86.7	8.1	`|`	12.3	0.1
`}`	78.9	7.4	`+`	3.8	0.1
,	13.9	6.1	`?:`	7.3	0.0
`:`	74.3	1.7	`?`	7.1	0.0
header-name	97.7	1.5	`do`	21.3	0.0
\\	100.0	0.9	`#error`	25.1	0.0
`#endif`	81.9	0.8	`:b`	7.2	0.0
`else`	42.2	0.7	`double`	3.1	0.0
string-literal	8.0	0.4	`^`	3.1	0.0
`void`	18.2	0.4	`union`	6.2	0.0
`&&`	17.8	0.2			

One way of differentiating multiline expressions is for the start, and end, of the lines to differ from other lines containing expressions. One possible way of differentiating the two ends of a line is to use tokens that don't commonly appear in those locations. For instance, lines often end in a semicolon, not an arithmetic operator (see Table 940.1), and at the start of a line additional indentation for the second and subsequent lines containing the same expression will set it off from the surrounding code. Some developers prefer to split expressions just before binary operators. However, the appearance of an operator as the last non-white-space character is more likely to be noticed than the nonappearance

of a semicolon (the human visual system is better at detecting the presence rather than the absence of a stimulus). Of course, the same argument can be given for an identifier or operator at the start of a line. These coding guidelines give great weight to existing practice. In this case this points to splitting expressions before/after binary operators; however, there is insufficient evidence of a worthwhile benefit for any guideline recommendation.

770 distinguishing features

Optimization

Many developers have a view of expressions that treats them as standalone entities. This viewpoint is often extended to translator behavior, which is then thought to optimize and generate machine code on an expression-by-expression basis. This developer though process leads on to the idea that performing as many operations as much as possible within a single expression evaluation results in translators generating more efficient machine code. This thought process is not cost effective because the difference in efficiency of expressions written in this way is rarely sufficient to warrant the cost, to the current author and subsequent readers, of having to comprehend them.

Whether a complex expression results in more, or less, efficient machine code will depend on the optimization technology used by the translator. Although modern optimization technology works on units significantly larger than an expression, there are still translators in use that operate at the level of individual expressions.

0 translator optimizations

Example

```
1  extern int g(void);
2  extern int a,
3             b;
4
5  void f(void)
6  {
7  a + b;           /* A computation. */
8  a;               /* An object. */
9  g();             /* A function. */
10 a = b;           /* Generates side effect. */
11 a = b , a + g(); /* A combination of all of the above. */
12 }
```

Usage

A study by Bodík, Gupta, and Soffa[131] found that 13.9% of the expressions in SPEC95 were partially redundant, that is, their evaluation is not necessary under some conditions.

190 partial redundancy elimination

See Table 1713.1 for information on occurrences of full expressions, and Table 770.2 for visual spacing between binary operators and their operands.

1712 full expression

Table 940.2: Occurrence of a token as the first token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). */* new-line */* denotes a comment containing one or more new-line characters, while */* ... */* denotes that form of comment on a single line. Based on the visible form of the `.c` files.

Token	% First Token on Line	% Occurrence of Token	Token	% First Token on Line	% Occurrence of Token
default	0.2	99.9	**volatile**	0.0	50.0
#	5.0	99.9	**int**	1.8	47.0
typedef	0.1	99.8	**unsigned**	0.7	46.8
static	2.1	99.8	**struct**	1.1	38.9
for	0.8	99.7	**const**	0.1	35.5
extern	0.2	99.6	**char**	0.5	30.5
switch	0.3	99.4	**void**	0.6	28.7
case	1.6	97.8	***v**	0.5	28.7
* new-line *\	13.7	97.7	**++v**	0.0	27.8
register	0.2	95.0	**signed**	0.0	27.2
return	3.3	94.5	**&&**	0.3	21.2
goto	0.4	94.1	identifier	31.1	20.8
if	6.9	93.6	**||**	0.2	18.4
break	1.2	91.8	**--v**	0.0	17.9
continue	0.2	91.3	**short**	0.0	16.0
}	8.3	88.3	**#error**	0.0	15.6
do	0.1	87.3	*string-literal*	0.6	12.4
while	0.4	85.2	**sizeof**	0.1	11.3
enum	0.1	73.7	**long**	0.1	10.1
\\	0.6	70.8	*integer-constant*	2.2	6.6
else	1.1	70.2	**?**	0.0	5.6
union	0.0	63.3	**&v**	0.1	5.2
/* ... *\	5.4	62.6	**-v**	0.1	5.0
{	5.1	54.9	**?:**	0.0	5.0
float	0.0	54.0	**|**	0.0	4.2
double	0.0	53.6	*floating-constant*	0.0	4.1

value profiling

Recent research[189,467,859] has found that for a few expressions, a large percentage of their evaluations return the same value during program execution. Depending on the expression context and the probability of the same value occurring, various optimizations become worthwhile[986] (0.04% of possible expressions evaluating to the same value a sufficient percentage of the time in a context that creates a worthwhile optimization opportunity). Some impressive performance improvements (more than 10%) have been obtained for relatively small numbers of optimizations. Citron[238] studied how processors might detect previously executed instruction sequences and reuse the saved results (assuming the input values were the same).

Table 940.3: Breakdown of invariance by instruction types. These categories include integer loads (*ILd*), floating-point loads (*FLd*), load address calculations (*LdA*), stores (*St*), integer multiplication (*IMul*), floating-point multiplication (*FMul*), floating-point division (*FDiv*), all other integer arithmetic (*IArth*), all other floating-point arithmetic (*FArith*), compare (*Cmp*), shift (*Shft*), conditional moves (*CMov*), and all other floating-point operations (*FOps*). The first number shown is the percent invariance of the topmost value for a class type, while the number in parenthesis is the dynamic execution frequency of that type. Results are not shown for instruction types that do not write a register (e.g., branches). Adapted from Calder, Feller, and Eustace.[189]

Program	ILd	FLd	LdA	St	IMul	FMul	FDiv	IArth	FArth	Cmp	Shft	CMov	F
compress	44(27)	0(0)	88(2)	16(9)	15(0)	0(0)	0(0)	11(36)	0(0)	92(2)	14(9)	0(0)	
gcc	46(24)	83(0)	59(9)	48(11)	40(0)	30(0)	31(0)	46(28)	0(0)	87(3)	54(7)	51(1)	9
go	36(30)	100(0)	71(13)	35(8)	18(0)	100(0)	0(0)	29(31)	0(0)	73(4)	42(0)	52(1)	10
ijpeg	19(18)	73(0)	9(11)	20(5)	10(1)	68(0)	0(0)	15(37)	0(0)	96(2)	17(21)	15(0)	9
li	40(30)	100(0)	27(8)	42(15)	30(0)	13(0)	0(0)	56(22)	0(0)	93(2)	79(3)	60(0)	10
perl	70(24)	54(3)	81(7)	59(15)	2(0)	50(0)	19(0)	65(22)	34(0)	87(4)	69(6)	28(1)	5
m88ksim	76(22)	59(0)	68(8)	79(11)	33(0)	53(0)	66(0)	64(28)	100(0)	91(5)	66(6)	65(0)	10
vortex	61(29)	99(0)	46(6)	65(14)	9(0)	4(0)	0(0)	70(31)	0(0)	98(2)	40(3)	20(0)	10

Studies of operand values during program execution (investigating ways of minimizing processor power consumption) have found that a significant percentage of these values use fewer representation bits than are available to them (i.e., they are small positive quantities). Brooks and Martonosi[163] found that 50% of operand values in SPECint95 required less than 16 bits.

Table 940.4: Number of objects defined (in a variety of small multimedia and scientific programs) to have types represented using a given number of bits (i.e., mostly 32-bit **int**) and number of objects having a maximum bit-width usage (i.e., number of bits required to represent any of the values stored in the object; rounded up to the nearest byte boundary). Adapted from Stephenson,[1290] whose analysis was performed by static analysis of the source.

Bits	Objects Defined	Objects Requiring Specified Bits
1	0	203
8	7	134
16	27	108
32	686	275

941 Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. DR287)

object modified once between sequence points

Commentary

A violation of this requirement results in undefined behavior. If an object is modified more than once between sequence points, the standard does not specify which modification is the last one. The situation can be even more complicated when the same object is read and modified between the same two sequence points. This requirement does not specify exactly what is meant by *object*. For instance, the following full expression may be considered to modify the object `arr` more than once between the same sequences points.

942 object read and modified between sequence points

```
1  int arr[10];
2
3  void f(void)
4  {
5  arr[1]=arr[2]++;
6  }
```

C++

Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. 5p4

The C++ Standard avoids any ambiguity in the interpretation of *object* by specifying scalar type.

Other Languages

In most languages assignment is not usually considered to be an operator, and assignment is usually the only operator that can modify the value of an object; other operators that modify objects are not often available. In such languages function calls is often the only mechanism for causing more than one modification between two sequence points (assuming that such a concept is defined, which it is not in most languages).

Common Implementations

Most implementations attempt to generate the best machine code they can for a given expression, independently of how many times the same object is modified. Since the surrounding context often has a strong influence on the code generated for an expression, it is possible that the evaluation order for the same expression will depend on the context in which it occurs.

Coding Guidelines

As the example below shows, a guideline recommendation against modifying the same object more than once between two adjacent sequence points is not sufficient to guarantee consistent behavior. A guideline recommendation that is sufficient to guarantee such behavior is discussed elsewhere.

expression 944.1
same result for all evaluation orders

Example

In following the first expression modifies `glob` more than once between sequence points:

```
1  extern int glob,
2             valu;
3
4  void f(void)
5  {
6  glob = valu + glob++;                    /* Undefined behavior. */
7  glob = (glob++, glob) + (glob++, glob); /* Undefined and unspecified behavior. */
8  }
```

Possible values for `glob`, immediately after the sequence point at the semicolon punctuator, include

- `valu + glob`
- `glob + 1`
- `((valu + glob) && 0xff00) | ((glob + 1) && 0x00ff)`

The third possibility assumes a 16-bit representation for **`int`**— a processor whose store operation updates storage a byte at a time and interleaves different store operations. In the second expression the evaluation of the left operand of the comma operator may be overlapped. For instance, a processor that has two arithmetic logic units may split the evaluation of an expression across both units to improve performance. In this case `glob` is modified more than once between sequence points. Also, the order of evaluation is unspecified.

expression 944
order of evaluation

In the following:

```
1  struct T {
2            int mem_1;
3            char mem_2;
4           } *p_t;
5
6  extern void f(int, struct T);
7
8  void g(void)
9  {
10 int loc = (*p_t).mem_1++ + (*p_t).mem_2++;
11         f((*p_t).mem_1++,   *p_t)        ; /* Modify part of an object. */
12 }
```

there is an object, `*p_t`, containing various subobjects. It would be surprising if a modification of a subobject (e.g., `(*p_t).mem_1`) was considered to be the same as a modification of the entire object. If it was, then the two modifications in the initialization of expression for `loc` would result in undefined behavior. In the call to `f` the first argument modifies a subobject of the object `*p_t`, while the second argument accesses all of the object `*p_t` (and undefined behavior is to be expected, although not explicitly specified by the standard).

Furthermore, the prior value shall be read only to determine the value to be stored.[71)] 942

object read and modified between sequence points

Commentary

In expressions, such as `i++` and `i = i*2`, the value of the object `i` has to be read before its value can be operated on and a potentially modified value written back. The semantics of the respective operators ensure that this ordering between operations occurs.

In expressions, such as `j = i + i--`, the object `i` is read twice and modified once. The left operand of the binary plus operator performs a read of `i` that is not necessary to determine the value to be stored into it. The behavior is therefore undefined. There are also cases where the object being modified occurs on the left side of an assignment operator; for instance, `a[i++] = i` contains two reads from `i` to determine a value and a modification of `i`.

Coding Guidelines

The generalized case of this undefined behavior is covered by a guideline recommendation dealing with evaluation order.

944.1 expression same result for all evaluation orders

---

943 The grouping of operators and operands is indicated by the syntax.[72)]

expression grouping operator precedence
943 operator precedence
955 operator associativity

Commentary

The two factors that control the grouping are precedence and associativity.

Other Languages

Most programming languages are defined in terms of some form of formal, or semiformal, BNF syntax notation. While a few languages allow operators to be overloaded, they usually keep their original precedence. In APL all operators have the same precedence and expressions are interpreted right-to-left (e.g., `1*2+3` is equivalent to `1*(2+3)`). The designers of Ada recognized[619] that developers do not have the same amount of experience handling the precedence of the logical operators as they do the arithmetic operators. An expression containing a sequence of the same logical binary operator need not be parenthesized, but a sequence of different logical binary operators must be parenthesized (parentheses are not required for unary **`not`**).

Common Implementations

Most implementations perform the syntax analysis using a table-driven parser. The tables for the parser are generated using some automatic tool (e.g., `yacc`, `bison`) that takes a LALR(1) grammar as input. The grammar, as specified in the standard, and summarized in Annex A, is not in LALR(1) form as specified. It is possible to transform it into this form, an operation that is often performed manually.

Coding Guidelines

Developers over learn various skills during the time they spend in formal education. These skills include the following:

- The order in which words are spoken is generally intended to reduce the comprehension effort needed by the listener. The written form of languages usually differs from the spoken form. In the case of English, it has been shown[1083] that readers parse its written form left-to-right, the order in which the words are written. It has not been confirmed that readers of languages written right-to-left parse them in a right-to-left order.
- Many science and engineering courses require students to manipulate expressions containing operators that also occur in source code. Students learn, for instance, that in an expression containing a multiplication and addition operator, the multiplication is performed first. Substantial experience is gained over many years in reading and writing such expressions. Knowledge of the ordering relationships between assignment, subtraction, and division also needs to be used on a very frequent basis. Through constant practice, knowledge of the precedence relationships between these operators becomes second nature; developers often claim that they are natural (they are not, it is just constant practice that makes them appear so).

Your author knows of no research studying how developers read expressions. The assumption made in these coding guidelines subsections is that developers' extensive experience reading prose is a significant factor affecting how they read source code. Given the significant differences in the syntactic structure of natural

770 reading practice

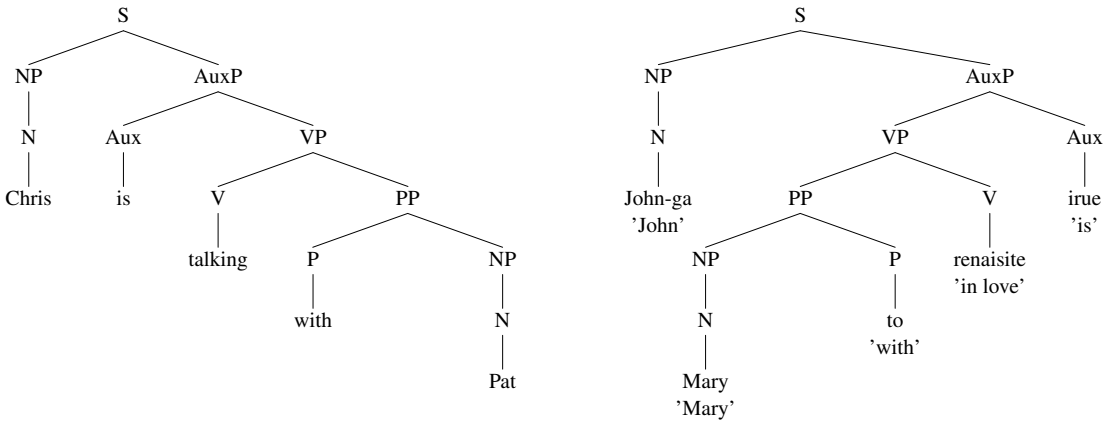


Figure 943.1: English ("Chris is talking with Pat") and Japanese ("John-ga Mary to renaisite irue") language phrase structure for sentences of similar complexity and structure. While the Japanese structure may seem back-to-front to English speakers, it appears perfectly natural to native speakers of Japanese. Adapted from Baker.[86]

languages (see Figure 943.1) the possibility of an optimal visual expression organization, which is universal to all software developers, seems remote.

While developers are likely to have had significant previous experience in reading and writing the commonly occurring C operators, few of developers attain the same level of practice with the other operators. Consequently, developer knowledge of the precedence levels of other operators is often faulty (which does not prevent them from being willing to jump to conclusions). A study by Jones[685] found that subjects (developers with an average of 14 years professional experience) failed to correctly parenthesize 33% of expressions containing two binary operators.

operator 770
relative spacing
operand 792
name context

Factors that have been found to effect developer operator precedence decisions include the relative spacing between operators and the names of the operands.

One solution to faulty developer knowledge of operator precedence levels is to require the parenthesizing of all subexpressions (rendering any precedence knowledge the developer may have, right or wrong, irrelevant). Such a requirement often brings howls of protest from developers. Completely unsubstantiated claims are made about the difficulties caused by the use of parentheses. (The typing cost is insignificant; the claimed unnaturalness is caused by developers who are not used to reading parenthesized expressions, and so on for other developer complaints.) Developers might correctly point out that the additional parentheses are redundant (they are in the sense that the precedence is defined by C syntax and the translator does not require them); however, they are not redundant for readers who do not know the correct precedence levels.

An alternative to requiring parentheses for any expression containing more than two operators is to provide a list of special where it is believed that developers are very unlikely to make mistakes (these cases have the advantage of being common). Listing special cases could either be viewed as the thin end of the edge that eventually drives out use of parentheses, or as an approach that gradually overcomes developer resistance to the use of parentheses.

When combined with binary operators, the correct order of evaluation of unary operators is simple to deduce and developers are unlikely to make mistakes in this case. However, the ordering relationship, when a unary operator is applied to the result of another unary operator, is easily confused when unary operators appear to both the left and right of the same operand. This is a case where the use of parentheses removes the possibility of reader mistakes.

In C both function calls and array indexing are classified as operators. There is likely to be considerable developer resistance to parenthesizing these operators because they are not usually thought of in these terms (they are not operators in many other languages); they are also unary operators and the pair of characters used is often considered as forming bracketed subexpressions.

In the following guideline recommendation the expression within

- the square brackets used as an array subscript operator are treated as equivalent to a pair of matching parentheses, not as an operator; and
- the arguments in a function invocation are each treated as full expressions and are not considered to be part of the rest of the expression that contains the function invocation for the purposes of the deviations listed.

An issue related to precedence, but not encountered so often, is associativity, which deals with the evaluation order of operands when the operators have the same precedence. If the operands in an expression have different types, the evaluation order specifies the pairings of operand types that need to go through the usually arithmetic conversions.

955 operator associativity

706 usual arithmetic conversions

Cg 943.1 Each subexpression of a full expression containing more than one operator shall be parenthesized.

Dev 943.1 A full expression that only contains zero or more additive operators and a single assignment operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more multiplication, division, addition, and subtraction operators and a single assignment operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more additive operators and a single relational or equality operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more multiplicative and additive operators and a single relational or equality operator need not be parenthesized.

Developers appear to be willing to accept the use of parentheses in so-called complex expressions. (An expression containing a large number of operators, or many different operators, is often considered complex; exactly how many operators is needed varies depending on who is asked.) Your author's unsubstantiated claim is that more time is spent discussing under what circumstances parentheses should be used than would be spent fully parenthesizing every expression developers ever write. Management needs to stand firm and minimize discussion on this issue.

Example

```
1  *p++;            /* Equivalent to *(p++); */
2
3  (char)!+-~***+p; /* Operators applied using an inside out order. */
4
5  ;m<-++pq++->m;   /* The token -> is not usually thought of as a unary operator. */
6
7  a = b = c;       /* Equivalent to  a = (b  = c); */
8  x + y + z;       /* Equivalent to (x +  y) + z ; */
```

944 Except as specified later (for the function-call `()`, `&&`, `||`, `?:`, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

expression order of evaluation

Commentary

The exceptional cases are all operators that involve a sequence point during their evaluation.

This specification, from the legalistic point of view, renders all expressions containing more than one operand as containing unspecified behavior. However, the definition of strictly conforming specifies that the output must not be dependent on any unspecified behavior. In the vast majority of cases all orders of evaluation of an expression deliver the same result.

91 strictly conforming program output shall not

Other Languages

Most languages do not define an order of evaluation for expressions. Snobol 4 defines a left-to-right order of evaluation for expressions. The Ada Standard specifies " . . . in some order that is not defined", with the intent[619] that there is some order and that this excludes parallel evaluation. Java specifies a left-to-right evaluation order. The left operand of a binary operator is fully evaluated before the right operand is evaluated.

Common Implementations

Many implementations build an expression tree while performing syntax analysis. At some point this expression tree is walked (often in preorder, sometimes in post-order) to generate a lower-level representation (sometimes a high-level machine code form, or even machine code for the executing host). An optimizer will invariably reorganize this tree (if not at the C level, then potentially though code motion of the intermediate or machine code form).

Even the case where a translator performs no optimizations and the expression tree has a one-to-one mapping from the source, it is not possible to reliably predict the order of evaluation. (There is more than one way to walk an expression tree matching higher-level constructs and map them to machine code.) As a general rule, increasing the number of optimizations performed increases the unpredictability of the order of expression evaluation.

Coding Guidelines

The order of evaluation might not affect the output from a program, but it can affect its timeliness. In:

```
1  printf("Hello ");
2  x = time_consuming_calculation() + print("World\n");
```

the order in which the two function calls on the right-hand side of the assignment are invoked will affect how much delay occurs between the output of the character sequences **Hello** and **World**.

In the expression `i = func(1) + func(2)`, the value assigned to `i` may, or may not, depend on the order in which the two invocations of `func` occur. Also the order of invocation may result in other objects having differing values. The sequence point that occurs prior to each function being invoked does not prevent these different behaviors from occurring. Sequence points are too narrow a perspective; it is necessary to consider the expression evaluation as a whole.

function call sequence point 1025

Cg 944.1

The state of the C abstract machine, after the evaluation of a full expression, shall not depend on the order of evaluation of subexpressions or the order in which side effects take place.

Example

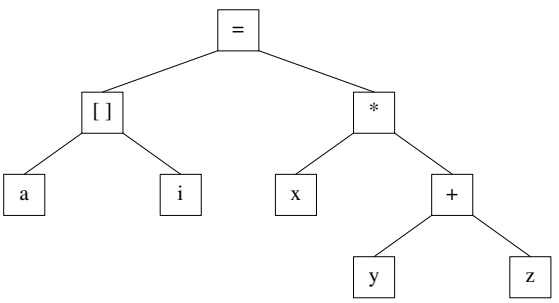


Figure 944.1: A simplified form of the kind of tree structure that is likely to be built by a translator for the expression `a[i]=x*(y+z)`.

```
1   #include <stdio.h>
2
3   extern volatile int glob;
4
5   void f(void)
6   {
7   int loc = glob + glob * glob;
8
9   /*
10   * In the following the only constraints on the order in
11   * which characters appear are that:
12   *    ) x must be output before y and
13   *    ) a must be output before b
14   */
15  loc = printf("x"),printf("y") + printf("a"),printf("b");
16  }
```

945 Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. bitwise operators

Commentary

This defines the term *bitwise operators*.

C++

The C++ Standard does not define the term bitwise operators, although it does use the term bitwise in the description of the **&**, **^** and **|** operators.

Other Languages

PL/1 has a bit data type and supports bitwise operations on values having such types.

Coding Guidelines

Bitwise operations provide a means for manipulating an object's underlying representation. They also provide a mechanism for using a new data type, the *bit-set*. There is a guideline recommendation against making use of an object's underlying representation. The following discussion looks at possible deviations to this recommendation. 569.1 representation information using

Performance issues

The result of some sequences of bitwise operations are the same as some arithmetic operations. For instance, left-shifting and multiplication by powers of two. There is a general belief among developers that processors execute these bitwise instructions faster than the arithmetic instructions. The extent to which this belief is true varies between processors (it tends to be greater in markets where processor cost has been traded-off against performance). The extent to which a translator automatically performs these mappings will depend on whether it has sufficient information about operand values and the quality of the optimizations it performs. If performance is an issue, and the translator does not perform the desired optimizations, the benefit of using bitwise operations may outweigh any other factors that increase costs, including:

- Subsequent reader comprehension effort— switching between thinking about bitwise and arithmetic operations will require at least a cognitive task switch. 0 cognitive switch
- The risk that a change of representation in the types used will result in the bitwise mapping used failing to apply. This may cause faults to occur.
- Treating the same object as having different representations, in different parts of the visible source requires readers to use two different mental models of the object. Two models may require more cognitive effort to recall and manipulate than one, and interference may also occur in the reader's memory, potentially leading to mistakes being made.

Dev 569.1 A program may use bitwise operators to perform arithmetic operations provided a worthwhile cost/benefit has been shown to exist.

Bit-set

Some applications, or algorithms, call for the creation of a particular kind of set data type (in mathematics a set can hold many values, but only one of each value). The term commonly used to describe this particular kind of set is *bit-set*, which is essentially an array of boolean values. The technique used to implement this bit-set type is to interpret every bit of an integer type as representing a member of the set. (When the bit is set, the member is considered to be in the set; when it is not set, the member is not present.) The number of members that can be represented using this technique is limited by the number of bits available in an integer type. This technique essentially provides both storage and performance optimization. An alternative representation technique is a structure type containing a member for each member of the bit-set, and appropriate functions for testing and setting these members.

boolean role 476

While the boolean role is defined in terms of operations that may be performed on a value having certain properties, it is possible to define a bit-set role in terms of the operations that may be performed on a value having certain properties.

bit-set role

An object having an integer type, or value having an integer type has a bit-set role if it appears as the operand of a bitwise operator or the object is assigned a value having a bit-set role.

bitwise operator result bit-set role numeric role

For the purpose of these guideline recommendations the result of a bitwise operator has a bit-set role.

An object having an integer type, or value having an integer type has a numeric role if it appears as the operand of an arithmetic operator or the object is assigned a value having a numeric role. Objects having a floating type always have a numeric role.

arithmetic operator result numeric role

For the purpose of these guideline recommendations the result of an arithmetic operator is defined to have a numeric role.

integer promotions 675
usual arithmetic conversions 706

The sign bit, if any, in the value representation shall not be used in representing a bit-set. (This restriction is needed because, if an operand has a signed type, the integer promotions or the usual arithmetic conversions can result in an increase in the number of bits used in the value representation.)

Dev 569.1 An object having a bit-set role that appears as the operand of a bitwise operator is not considered to be making use of representation information.

Example

Bitwise operations allow several conditions to be checked at the same time.

```
#define R_OK (0x01)
#define W_OK (0x02)
#define X_OK (0x04)

_Bool f(unsigned int permission)
{
return (permission & (R_OK | W_OK)) == 0;
}
```

---

bitwise operations signed types

These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types. 946

Commentary

The choice of behavior was largely influenced by what the commonly available processors did at the time the standard was originally written. In some cases there is a small set of predictable behaviors; for instance, left-shift can exhibit undefined behavior, while under the same conditions right-shift is implementation-defined.

left-shift 1193
undefined
right-shift 1196
negative value

Efficiency of execution has been given priority over specifying the exact behavior (which may be inefficient to implement on some processors).

Warren[1443] provides an extensive discussion of calculations that can be performed and information obtained via bitwise operations on values represented in two's complement notation.

C++

These operators exhibit the same range of behaviors in C++. This is called out within the individual descriptions of each operator in the C++ Standard.

Other Languages

The issues involved are not specific to C. They are caused by the underlying processor representations of integers and how instructions that perform bitwise operations on these types are defined to operate. As such, other languages that support bitwise operations also tend to exhibit the same kinds of behaviors.

Coding Guidelines

The issues involved in using operators that rely on undefined and implementation-defined behavior are discussed under the respective operators.

---

947 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined. exception condition

Commentary

This defines the term *exceptional condition*. Note that the wording does not specify that an exception is raised, but that the condition is exceptional (i.e., unusual). These exceptional conditions can only occur for operations involving values that have signed integer types or real types.

There are only a few cases where results are not mathematically defined (e.g., divide by zero). The more common case is the mathematical result not being within the range of values supported by its type (a form of overflow). For operations on real types, whether values such as infinity or NaN are representable will depend on the representation used. In the case of IEC 60559 there is always a value that is capable of representing the result of any of its defined operations.

C90

The term *exception* was defined in the C90 Standard, not *exceptional condition*.

C++

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression is a constant expression (5.19), in which case the program is ill-formed. 5p5

The C++ language contains explicit exception-handling constructs (Clause 15, **try**/**throw** blocks). However, these are not related to the mechanisms being described in the C Standard. The term exceptional condition is not defined in the C sense.

Other Languages

Few languages define the behavior when the result of an expression evaluation is not representable in its type. However, Ada does define the behavior— it requires an exception to be raised for these cases.

Common Implementations

In most cases translators generate the appropriate host processor instruction to perform an operation. Whatever behavior these instructions exhibit, for results that are not representable in the operand type, is the implementation's undefined behavior. For instance, many processors trap if the denominator in a division operation is zero. It is rare for an implementation to attempt to detect that the result of an expression

evaluation overflows the range of values representable in its type. Part of the reason is efficiency and part because of developer expectations (an implementation is not expected to do it).

On many processors the instructions performing the arithmetic operations are defined to set a specified bit if the result overflows. However, the unit of representation is usually a register (some processors have instructions that operate on a subdivision of a register— a halfword or byte). For C types that exactly map to a processor register, detecting an overflow is usually a matter of generating an additional instruction after every arithmetic operation (branch on overflow flag set). Complications can arise for mixed signed/unsigned expressions if the processor also sets the overflow flag for operations involving unsigned types. (The Intel x86, IBM 370 set the carry flag in this case; SPARC has two add instructions, one that sets the carry flag and one that does not.) A few processors have versions of arithmetic instructions that are either defined to trap on overflow (often limited to add and subtract, e.g., MIPS) or provide a mechanism for toggling trap on overflow (IBM 370, HP–was DEC– VAX).

Example

In the following the multiplication by `LONG_MAX` will deliver a result that is not representable in a **`long`**.

```
1  extern int i;
2  extern long j;
3
4  void f(void)
5  {
6  i = 30000;
7  j = i * LONG_MAX;
8  }
```

---

effective type

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[73)] 948

Commentary

This defines the term *effective type*, which was introduced into C99 to deal with objects having allocated storage duration. In particular, to provide a documented basis for optimizers to attempt to work out which objects might be aliased, with a view to generating higher-quality machine code. Knowing that a referenced object is not aliased at a particular point in the program can result in significant performance improvements (e.g., it might be possible to deduce that its value can be held in a register throughout the execution of a critical loop rather than loaded from storage on every iteration).

Computing alias information can be very resource (processor time and storage needed) intensive. To reduce this overhead, translator vendors try to make simplifying assumptions. One assumption commonly made is that pointers to `type_A` are disjoint from pointers to `type_B`. The concept of effective type provides a mechanism for knowing the possible types that an object can be referenced through. If the same object is accessed using effective types that do not meet the requirements specified in the standard the behavior is undefined; one possible behavior is to do what an optimizing translator happens to do based on the assumption that accesses through different effective types do not occur.

object 960 value accessed if type

Storing a value into an object that has a declared type, through an lvalue having a different type, does not change that object's effective type.

C90

The term *effective type* is new in C99.

C++

The term *effective type* is not defined in C++. A type needs to be specified when the C++ new operator is used. However, the C++ Standard includes the C library, so it is possible to allocate storage via a call to the `malloc` library function, which does not associate a type with the allocated storage.

Common Implementations

The RTC tool[866] performs type checking on accesses to objects during program execution. The type information associated with every storage location written to specifies the number of bytes in the type and one of unallocated, uninitialized, integer, real, or pointer. The type of a write to a storage location is checked against the declared type of that location, if any, and the type of a read from a location is checked against the type of the value last written to it.

Coding Guidelines

While an understanding of effective type might be needed to appreciate the details of how library functions such as `memcpy` and `memcmp` operate, developers rarely need to get involved in this level of detail.

---

949 If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value.

Commentary

Only objects with allocated storage duration have no declared type. The type is assigned to such an object through a value being stored into it in name only; there is no requirement for this information to be represented during program execution (although implementations designed to aid program debugging sometimes do so). The type of an object with allocated storage duration is potentially changed every time a value is stored into it. A parallel can be drawn between such an object and another one having a union type.

Storing a value through an lvalue occurs when the left operand of an assignment operator is a dereferenced pointer value. The effective type is derived from the dereferenced pointer type in this case.

The character types are special in that they are the types often used to access the individual bytes in an object (e.g., to copy an object). This usage is sufficiently common that the Committee could not mandate that an object modified via an lvalue having a character type will only be accessed via a character type (it would also create complications for the specification of some of the library functions— e.g., `memcpy`.) An object having allocated storage duration can only have a character type as its effective type if it is accessed using such a type.

959 effective type
lvalue used for access

Other Languages

Many languages that support dynamic storage allocation require that a type be associated with that allocated storage. Some languages (e.g., awk) allocate storage implicitly without the need for any explicit operation by the developer.

Coding Guidelines

Objects with no declared type must have allocated storage duration and can only be referred to via pointers (this C sentence refers to the effective type of the objects, not the type of the pointers that refer to them). Objects having automatic and static storage duration have a fixed effective type— the one appearing in their declaration. The type of an object having allocated storage duration can change every time a new assignment is made to it.

Allocating storage for an object and treating it as having `type_a` in one part of a program and later on treating it as having `type_b` creates a temporal dependency (the two kinds of usage have to be disjoint) and a spatial dependency (the allocated storage needs to be large enough to be able to represent both types). Keeping track of these dependencies is a cost (developer cognitive resources needed to learn, keep track of, and take them into account) that is often significantly greater than the benefit (smaller, slightly faster-executing program image through not deallocating and reallocating storage). Explicitly deallocating storage when it is not needed and allocating it when it is needed is a minor overhead that creates none of these dependencies between different parts of a program.

Having the same allocated object referred to by pointers of different types creates a union type in all but name:

```
1  #include <stdlib.h>
2
3  float *p_f;
4  int *p_i;
5
6  void f(void)
7  {
8  void *p_v = malloc(sizeof(float)); /* Assume float & int are same size. */
9
10 /* Treat as union. */
11 p_f = p_v;
12 p_i = p_v;
13
14 p_v = malloc(sizeof(float)+sizeof(int));
15
16 /* Treat as struct. */
17 p_f = p_v;
18 p_i = (int *)((char *)p_v + sizeof(float));
19 }
```

Cg 949.1

Once an object having no declared type is given an effective type, it shall not be given another effective type that is incompatible with the one it already has.

Dev 949.1

Any object having no declared type may be accessed through an lvalue having a character type.

---

footnote 71

71) This paragraph renders undefined statement expressions such as 950

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

Commentary

The phrase *statement expressions* is used to make a distinction between the full expression contained within the statement and the syntactic construct *expression-statement*. Expressions can exhibit undefined behavior, but statements cannot (or at least are not defined by the standard to do so).

Other Languages

Even languages that don't contain the ++ operator can exhibit undefined behavior for one of these cases. If a ++ operator is not available, a function may be written by the developer to mimic it (e.g., `a[post_inc(i)] := i`). Many languages do not define the order in which the evaluation of the operands in an assignment takes place, while a few do.

---

footnote 72

72) The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. 951

Commentary

Every operator is assigned a *precedence* relative to the other operators. When an operand syntactically appears between two operators, it binds to the operator with highest precedence. In C there are thirteen levels of precedence for the binary operators and three levels of precedence for the unary operators.

Requirements on the operands of operators, and their effects, appear in the constraints and semantics subclauses. These occur after the corresponding syntax subclause.

Other Languages

Many other language specification documents use a similar, precedence-based, section ordering. Ada has six levels of precedence, while operators in APL and Smalltalk all have the same precedence (operator/operand binding is decided by associativity).

Example

In the expression `a+b*c` multiply has a higher precedence and the operand `b` is operated on by it rather than the addition operator.

---

952 Thus, for example, the expressions allowed as the operands of the binary `+` operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6.

Commentary

The subsections occur in the standard in precedence order, highest to lowest. For instance, in `a + b*c` the result of the multiplicative operator (discussed in clause 6.5.5) is an operand of the additive operator (discussed in clause 6.5.6). Also the ordering of subclauses within a clause follows the ordering of the nonterminals listed in that syntax clause.

---

953 The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.5.1), subscripting brackets `[]` (6.5.2.1), function-call parentheses `()` (6.5.2.2), and the conditional operator `?:` (6.5.15).

Commentary

A *`cast-expression`* is a separate subclause because there is a context where this unary operator is not permitted to occur syntactically as the last operator operating on the left operand of an assignment operator (although some implementations support this usage as an extension). In this context a *`unary-expression`* is required.

1133 cast-expression syntax
1288 assignment-expression syntax
1080 unary-expression syntax

The parentheses `()`, subscripting brackets `[]`, and function-call parentheses `()` all provide a method of enclosing an expression within a bracketing construct that cuts it off from the syntactic effects of any surrounding operators. The conditional operator takes three operands, each of which are different syntactic expressions.

1264 conditional-expression syntax

Other Languages

Many languages do not consider array subscripting and function-call parentheses as operators.

---

954 Within each major subclause, the operators have the same precedence.

Commentary

However, the operators may have different associativity.

955 operator associativity

C++

This observation is true in the C++ Standard, but is not pointed out within that document.

Other Languages

Many language specification documents are similarly ordered.

---

955 Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

operator associativity

Commentary

Every binary operator is specified to have an *associativity*, which is either to the left or to the right. In C the assignment operators and the conditional ternary operators associate to the right; all other binary operators associate to the left. Associativity controls how operators at the same precedence level bind to their operands. Operators with left-associativity bind to operands from left-to-right, Operators with right-associativity bind from right-to-left.

943 operator precedence

Most syntax productions for C operators follow the pattern $X_n \Rightarrow X_n\, op\, X_{n+1}$ where X_n is the production for the operator, op, having precedence n (i.e., they associated to the left); for instance, `i / j / k` is equivalent to `(i / j) / k` rather than `i / (j / k)`. The pattern for *`conditional-expression`* (and similarly for *`assignment-expression`*) is $X_n \Rightarrow X_{n+1}?X_{n+1} : X_n$ (i.e., it associates to the right); for instance, `a ? b : c ? d : e` is equivalent to `a ? b : (c ? d : e)` rather than `(a ? b : c) ? d : e`.

Other Languages

Most algorithmic languages have similar associativity rules to C. However, operators in APL always associate right-to-left.

Coding Guidelines

Like precedence, possible developer misunderstandings about how operators associate can be solved using parentheses. Expressions, or parenthesized expressions that consist of a sequence of operators with the same precedence, might be thought to be beyond confusion. If the guideline recommendation specifying the use of parentheses is followed, associativity will not be a potential source of faults. However, some of the deviations for that guideline recommendation allow consideration for multiplicative operators to be omitted from the enforcement of the guideline. For the case of adjacent multiplicative operators, this deviation should not be applied.

expression shall be parenthesized 943.1

Cg 955.1

If the result of a multiplicative operator is the immediate operand of another multiplicative operator, then the two operators shall be separated by at least one parenthesis in the source.

If an expression consists solely of operations involving the binary plus operator, it might be thought that the only issue that need be considered, when ordering operands, is their values. However, there is a second issue that needs to be considered— their type. If the operand types are different, the final result can depend on the order in which they were written (which defines the order in which the usual arithmetic conversions are applied).

usual arithmetic conversions 706

Cg 955.2

If the result of an additive operator is the immediate operand of another additive operator, and the operands have different promoted types, then the two operators shall be separated by at least one parenthesis in the source.

Example

In the following the fact that `j` is added to `i` before `k` is added to the result is not of obvious interest until it is noticed that their types are all different.

```
1  extern float i;
2  extern short j;
3  extern unsigned long k;
4
5  void f(void)
6  {
7  int x, y;
8
9  x = i + j + k;
10
11 y = i / j / k; /* / associates to the left: (i / j) / k */
12
13 i /= j /= k;   /* /= associates to the right: i /= (j /= k) */
14 }
```

Associativity requires that `j` be added to `i`, after being promoted to type **`float`**. The result type of `i+j` (**`float`**) causes `k` to be converted to **`float`** before it is added. The sequence of implicit conversions would

have been different had the operators associated differently, or the use of parentheses created a different operand grouping. Dividing `i` by `j`, before dividing the result by `k`, gives a very different answer than dividing `i` by the result of dividing `j` by `k`.

---

956 73) Allocated objects have no declared type. footnote 73

Commentary

The library functions that create such objects (`malloc` and `calloc`) are declared to return the type pointer to **`void`**.

C90

The C90 Standard did not point this fact out.

C++

The C++ operator **`new`** allocates storage for objects. Its usage also specifies the type of the allocated object. The C library is also included in the C++ Standard, providing access to the `malloc` and `calloc` library functions (which do not contain a mechanism for specifying the type of the object created).

Other Languages

Some languages require type information to be part of the allocation request used to create allocated objects. The allocated object is specified to have this type. Other languages provide library functions that return the requested amount of storage, like C.

---

957 DR287) A floating-point status flag is not an object and can be set more than once within an expression. footnote DR287

Commentary

Processors invariably set various flags after each arithmetic operation, be it floating-point or integer. For instance, in `w*x + y*z` after each multiplication flags status flags denoting *result is zero* or *result overflows* may be set. Floating-point status flags differ from integer status flags in that Standard library functions are available for accessing and setting their value, which makes visible the order in which operations take place.

Implementations that support floating-point state are required to treat changes to it as a side-effect. But, by not treating floating-point status flags as an object, the undefined behavior that occurs when the same object is modified between sequence points does not occur. 199 side effect floating-point state 941 object modified once between sequence points

This footnote was added by the response to DR #287.

Example

```
1  /*
2   * set/clear or clear/set one of the floating-point exception flags:
3   */
4  (feclearexcept)(FE_OVERFLOW) + (feraiseexcept)(FE_OVERFLOW);
```

---

958 If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one.

Commentary

In the declarations of the library functions `memcpy` and `memmove`, the pointers used to denote both the object copied to and the object copied from have type pointer to **`void`**. There is insufficient information available in either of the declared parameter types to deduce an effective type. The only type information available is the effective type of the object that is copied. Another case where the object being copied would not have an

effective type, is when it is storage returned from a call to the `calloc` function which has not yet had a value of known effective type stored into it.

Here the effective type is being treated as a property of the object being copied from. Once set it can be carried around like a value. (From the source code analysis point of view, there is no requirement that this information be represented in an object during program execution.)

Use of character types to copy one object to another object is a common idiom. Some developers write their own object copy functions, or simply use an inline loop (often with the mistaken belief of improved efficiency or reduced complexity). The usage is sufficiently common that the standard needs to take account of it.

Other Languages

Many languages only allow object values to be copied through the use of an assignment statement. Few languages support pointer arithmetic (the mechanism needed to enable objects to be copied a byte at a time). While many language implementations provide a mechanism for calling functions written in C, which provides access to functions such as `memcpy`, they do not usually provide any additional specifications dealing with object types.

In some languages (e.g., awk, Perl) the type of a value is included in the information represented in an object (i.e., whether it is an integer, real, or string). This type information is assigned along with the value when objects are assigned.

Common Implementations

There are a few implementations that perform localized flow analysis, enabling them to make use of effective type information (even in the presence of calls to library functions). While performing full program analysis is possible in theory, for nontrivial programs the amount of storage and processor time required is far in excess of what is usually available to developers. There are also implementations that perform runtime checks based on type information associated with a given storage location.[866]

A few processors tag storage with the kinds of value held in it[1389] (e.g., integer or floating-point). These tags usually represent broad classes of types such as pointers, integers, and reals. This functionality might be of use to an implementation that performs runtime checks on executing programs, but is not required by the C Standard.

Example

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  void *obj_copy(void *obj_p, size_t obj_size)
5  {
6  void *new_obj_p = malloc(obj_size);
7
8  /*
9   * It would take some fancy analysis to work out, statically,
10  * the effective type of the object being copied here.
11  */
12 memcpy(new_obj_p, obj_p, obj_size);
13
14 return new_obj_p;
15 }
```

effective type
lvalue used for access

For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access. 959

Commentary

This is the effective type of last resort. The only type available is the one used to access the object. For instance, an object having allocated storage duration that has only had a value stored into it using lvalues of character type will not have an effective type. This wording does not specify that the type used for the access is the effective type for subsequent accesses, as it does in previous sentences.

Coding Guidelines

The question that needs to be asked is why the object being accessed does not have an effective type. An access to the storage returned by the `calloc` function before another value is assigned to it, is one situation that can occur because of the way a particular algorithm works. Unless the access is via an lvalue having a character type, use is being made of representation information; this is discussed elsewhere.

569.1 representation information using

---

960 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[74)]

object value accessed if type

Commentary

This list is sometimes known as the *aliasing rules* for C. Any access to the stored value of an object using a type that is not one of those listed next results in undefined behavior. To access the same object using one of the different types listed requires the use of a pointer type. Reading from a different member of a union type than the one last stored into is unspecified behavior.

589 union member when written to

The standard defines various cases where types have the same representation and alignment requirements, they all involve either signed/unsigned versions of the same integer type or qualified/unqualified versions of the same type. The intent is to allow objects of these types to interoperate. These cases are reflected in the rules listed in the following C sentences. There are also special access permissions given for the type **`unsigned char`**.

486 signed integer corresponding unsigned integer
495 positive signed integer type subrange of equivalent unsigned type
556 qualifiers representation and alignment
573 value copied using unsigned char

C90

In the C90 Standard the term used in the following types was *derived type*. The term *effective type* is new in the C99 Standard and is used throughout the same list.

Other Languages

Most typed languages do not allow an object to be accessed using a type that is different from its declared type. Accessing the stored value of an object through different types requires the ability to take the addresses of objects or to allocate untyped storage. Only a few languages offer such functionality.

Common Implementations

The only problem likely to be encountered with most implementations, in accessing the stored value of an object, is if the object being accessed is not suitably aligned for the type used to access it.

39 alignment

Coding Guidelines

The guideline recommendation dealing with the use of representation information may be applicable here.

569.1 representation information using

Example

The following is a simple example of the substitutions that these aliasing rules permit:

```
1   extern int glob;
2   extern double f_glob;
3
4   extern void g(int);
5
6   void f(int *p_1, float *p_2)
7   {
8   glob = 1;
9   *p_1 = 3;            /* May store value into object glob. */
10  g_1(glob);           /* Cannot replace the argument, glob, with 1. */
11
```

```
12  glob = *p_1;
13  *p_2 = f_glob * 8.6; /* Undefined behavior if store modifies glob. */
14  g_2(glob);           /* Translator can replace the argument, glob, with 2. */
15  }
```

Things become more complicated if an optimizer attempts to perform statement reordering. Moving the generated machine code that performs floating-point calculations to before the assignment to `glob` is likely to improve performance on pipelined processors. Alias analysis suggests that the objects pointed to by `p_1` and `p_2` must be different and that statement reordering is possible (because it will not affect the result). As the following invocation of `f` shows, this assumption may not be true.

alias analysis 1491

```
1  union {
2         int i;
3         float f;
4        } u_g;
5
6  void h(void)
7  {
8  f(&u_g.i, &u_g.f);
9  }
```

---

object stored value accessed only by

— a type compatible with the effective type of the object,

compatible type 632 additional rules

Commentary

Accessing an object using a different, but compatible type(i.e., an enumerated type and its compatible integer type) is thus guaranteed to deliver the same result.

C++

3.10p15

— *the dynamic type of the object,*

1.3.3 dynamic type

the type of the most derived object (1.8) to which the lvalue denoted by an lvalue expression refers. [Example: if a pointer (8.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (clause 10), the dynamic type of the expression `*p` is "`D`." References (8.3.2) are treated similarly.] The dynamic type of an rvalue expression is its static type.

The difference between an object's dynamic and static type only has meaning in C++.

compatible type 631 if

Use of effective type means that C gives types to some objects that have no type in C++. C++ requires the types to be the same, while C only requires that the types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

Coding Guidelines

The two objects having compatible types might have been declared using one or more typedef names, which may depend on conditional preprocessing directives. Ensuring that such types remain compatible is a software engineering issue that is outside the scope of these coding guidelines.

enumeration 1447 type compatible with

The issue of making use of enumerated types and the implementation's choice of compatible integer type is discussed elsewhere.

Example

```
1   extern int f(int);
2
3   void DR_053(void)
4   {
5   int (*fp1)(int) = f;
6   int (**fpp)() = &fp1;
7
8   /*
9    * In the following call the value of fp1 is being accessed by an
10   * lvalue that is different from its declared type, but is compatible
11   * with its effective type: (int (*)()) vs. (int (*)(int)).
12   */
13  (**fpp)(3);
14  }
```

962 — a qualified version of a type compatible with the effective type of the object,

Commentary

Qualification does not alter the representation or alignment of a type (or of pointers to it), only the translation-time semantics. Adding qualifiers to the type used to access the value of an object will not alter that value. The **volatile** qualifier only indicates that the value of an object may change in ways unknown to the translator (therefore the quality of generated machine code may be degraded because a translator cannot make use of previous accesses to optimize the current access).

556 qualifiers representation and alignment
746 pointer converting qualified/unqualified

Other Languages

Languages containing a qualifier that performs a function similar to the C **const** qualifier (i.e., a read-only qualifier) usually allow objects having that type to access other objects of the same, but unqualified, type.

Example

```
1   extern int glob;
2
3   void f(const int *p_i)
4   {
5   /*
6    * Only ever read the value pointed to by p_i, but may
7    * directly, or indirectly, cause glob to be modified.
8    */
9   }
10
11  void g(void)
12  {
13  const int max = 33;
14
15  f(&max);
16  f((const int *)&glob);
17  }
```

963 — a type that is the signed or unsigned type corresponding to the effective type of the object,

Commentary

The signed/unsigned versions of the same type are specified as having the same representation and alignment requirements to support this kind of access. The standard places no restriction here on the values represented by the stored value being accessed. The intent of this list is to specify possible aliasing circumstances, not possible behaviors.

footnote 509 31
footnote 971 74

Other Languages

Few languages support an unsigned type. Those that do support such a type do not require implementations to support the inter-accessing of signed and unsigned types of the form available in C.

Coding Guidelines

The range of nonnegative values of a signed integer type is required to be a subrange of the corresponding unsigned integer type. However, it cannot be assumed that this explicit permission to access an object using either a signed or unsigned version of its effective type means that the behavior is always defined. The guideline recommendation on making use of representation information is applicable here.

positive 495 signed integer type
subrange of equivalent unsigned type
representation information using 569.1
footnote 509 31

If an argument needs to be passed to a function accepting a pointer to the oppositely signed type, an explicit cast will be needed. The issues involved in such casts are discussed elsewhere.

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object, 964

Commentary

This is the combination of the previous two cases.

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or 965

Commentary

A particular object may be an element of an array or a member of a structure or union type. Objects having one of these derived types can be accessed as a whole; for instance, using an assignment operator (the array object will need to be a member of a structure or union type). It is this access as a whole that in turn accesses the stored value(s) of the members.

Common Implementations

A great deal of research has been invested in analyzing the pattern of indexes into arrays within loops, with a view to parallelizing the execution of that loop. But, for array objects outside of loops, relatively little research effort has been invested in attempting to track the contents of particular array's elements. There are a few research translators that break structure and union objects down into their constituent members when performing flow analysis. This enables a much finer-grain analysis of the aliasing information.

data dependency 988
array element held in register 1369

Example

```
#include <string.h>

extern long *p_l;

union U {
          int i;
          long l;
         } uil;
struct S {
           int i;
           long l;
          } sil_1,
            sil_2;
```

```
14  long a_l[3];
15
16  void f(void)
17  {
18  *p_l = 33;
19
20  /*
21   * The following four statements may all cause the value
22   * pointed to by p_l to be modified.
23   */
24  memset(&uil, 0, sizeof(uil));
25  memset(&sil, 0, sizeof(sil_1));
26  memset(&a_l, 0, sizeof(a_l));
27  sil_2 = sil_1;
28
29  if (*p_l == 33) /* This test is not guaranteed to be true. */
30     a_l[0] = 9;
31  }
```

966 — a character type.

Commentary

Prior to the invention of the **void** type (during the early evolution of C[1180]), pointer to character types were used as the generic method of passing values having different kinds of pointer types as arguments to function calls.

Although library functions have always been available for copying any number of bytes from one object to another (e.g., `memcpy`), many developers have preferred to perform inline copying (writing the loop at the point of copy) or to call their own functions. These preferences show no signs of dying out and the standard needs to continue to support the possibility of objects having character types being aliases for objects of other types.

C++

— *a* **char** *or* **unsigned char** *type.* 3.10p15

The C++ Standard does not explicitly specify support for the character type **signed char**. However, it does specify that the type **char** may have the same representation and range of values as **signed char** (or **unsigned char**). 516 char range, representation and behavio

It is common practice to access the subcomponents of an object using a **char** or **unsigned char** type. However, there is code that uses **signed char**, and it would be a brave vendor whose implementation did not assume that objects having type **signed char** were not a legitimate alias for accesses to any object.

Other Languages

While other languages may not condone the accessing of subcomponents of an object, their implementations sometimes provide mechanisms for making such accesses at the byte level.

Coding Guidelines

Accessing objects that do not have a character type, using an lvalue expression that has a character type is making use of representation information, which is covered by a guideline recommendation. The special case of the type **unsigned char** is discussed elsewhere. 569.1 representation information 573 value copied using unsigned char

967 A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[75)] contracted

Commentary

This defines the term *contracted*.

Some processors have instructions that perform more than one operation before delivering a result. The most commonly seen instance of such a multiple operation instruction is the multiply/add pair; taking three operands and delivering the result of evaluating `x + y * z`. This so-called *fused* multiply/add instruction reflects the kinds of operations commonly seen in numerical computations— for instance, matrix multiple and FFT calculations. A fused instruction may execute more quickly than the equivalent two separate instructions and may return a result of greater accuracy (because there are no conversions or rounding performed on the intermediate result).

This wording in the standard explicitly states that the use of such fused instructions is permitted (subject to the use of the `FP_CONTRACT` pragma) by the C Standard, even if it means that the final result of an expression is different from what it would have been had several independent instructions been used.

C90

This explicit permission is new in C99.

C++

The C++ Standard, like C90, is silent on this subject.

Other Languages

Very few languages get involved in the instruction level processor details when specifying the behavior of programs. Fortran does not explicitly mention contraction but some implementations make use of it.

Common Implementations

Some implementations made use of fused multiply/add instructions in their implementation of C90.

Coding Guidelines

An expression that is contracted by an implementation may be thought to deliver the double advantage of faster execution and greater accuracy. However, in some cases the accuracy of the complete calculation may be decreased. The issues associated with contracting an expression are discussed elsewhere.

contraction 974 undermine predictability

The **`FP_CONTRACT`** pragma in **`<math.h>`** provides a way to disallow contracted expressions. 968

Commentary

The `FP_CONTRACT` pragma also provides a way to allow contracted expressions if they are supported by the implementation.

C90

Support for the `FP_CONTRACT` pragma is new in C99.

C++

Support for the `FP_CONTRACT` pragma is new in C99 and not specified in the C++ Standard.

---

contracted how implementation-defined

Otherwise, whether and how expressions are contracted is implementation-defined.[76)] 969

Commentary

Contraction requires support from both the processor and the translator. (Even although fused instructions may be available on a processor, a translator may not provide the functionality needed to make use of them.) For instance, in `a*b+c*d` there are a number of options open to an implementation that supports a contracted multiply/add. The instruction sequence used to evaluate this expression is likely to be affected by the values from previous operations currently available in registers.

C++

The C++ Standard does not give implementations any permission to contract expressions. This does not mean they cannot contract expressions, but it does mean that there is no special dispensation for potentially returning different results.

Common Implementations

The operator combination multiply/add is the most commonly supported by processors because of the frequency of occurrence of this pair in FFT and matrix operations (these invariably occur in signal processing applications). Other forms of contraction have been proposed for other specialist applications (e.g., cryptography[1484]).

The floating-point units in the Intel i860[624] can operate in pipelined or scalar mode, with a variety of options on how the intermediate results are fed into the different units. Depending on the generated code it is possible for the evaluation of `a*b+z` to differ from `c*d+z`, even when the products `a*b` and `c*d` are equal (this issue is discussed in WG14 document N291).

Coding Guidelines

Even in those cases where a developer is aware that expression contraction may occur, there is no guarantee that it will be possible to estimate its impact. For complex expressions the implementation-defined behavior may be sufficiently complex that developers may have difficulty deducing which, if any, subexpression evaluations have been contracted. (One way of finding out the translator's behavior is to examine a listing of the generated machine code.) Once known, what use is this information, on contracted expressions, to a developer? Probably none. The developer needs to look at the issue from a less-detailed perspective.

The only rationale for supporting contracted expressions is improved runtime performance. In those situations where the possible improvement in performance offered by contraction is not required it only introduces uncertainty, a cost for no benefit. Because the default behavior is implementation-defined (no contraction unless requested might have been a better default choice by the C Committee), it is necessary for the developer to ensure that contraction is explicitly switched off, unless it is explicitly required to be on.

Rev 969.1

Unless there is a worthwhile cost/benefit in allowing translators to perform contraction, any source file that evaluates floating-point expressions shall contain the preprocessing directive:
`#pragma STDC FP_CONTRACT off`
near the start of the source file, before the translation of any floating-point expressions.

When using the `FP_CONTRACT` pragma developers might choose to minimize the region of source code over which it is in the "ON" state (i.e., having a matching pragma directive that switches it to the "OFF" state) or have it the "ON" state during the translation of an entire translation unit. Until more experience is gained with the use of `FP_CONTRACT` pragma it is not possible to evaluate whether any guideline recommendation is worthwhile.

---

970 **Forward references:** the **`FP_CONTRACT`** pragma (7.12.2), copying functions (7.21.2).

---

971 74) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

footnote 74 object aliased

Commentary

An object may be aliased under other circumstances, but the standard does not require an implementation to support any other circumstances. Aliasing is discussed in more detail in the discussion of the **`restrict`** type qualifier.

1491 alias analysis

Other Languages

The potential for aliasing is an issue in the design of most programming languages, although this term may not explicitly appear in the language definition. There is a family of languages having the major design aim of preventing any aliasing from occurring— functional languages.

Coding Guidelines

Although some coding guideline documents warn about the dangers of creating aliases (e.g., developers need to invest effort in locating, remembering, and taking them into account), their cost/benefit in relation to alternative techniques (e.g., moving the declaration of an object from block to file scope rather than passing

its address as an argument in what appears to be a function call) is often difficult to calculate (experience suggest that developers rarely create aliases unless they are required). Given the difficulty of calculating the cost/benefit of various alternative constructs these coding guidelines are silent on the issue of alias creation.

```
1  extern int glob;
2
3  int f(int *valu)
4  {
5  return ++(*valu) + glob; /* Can valu ever refer to glob? */
6  }
```

---

footnote 75

75) A contracted expression might also omit the raising of floating-point exceptions. 972

Commentary

exception 947
condition
NaN 340
raising an exception

For instance, an exception might be raised when the evaluation of an expression is not mathematically defined, or when an operand has a NaN value. To obtain the performance improvement implied by fused operations, a processor is likely to minimize the amount of checking it performs on any intermediate results. Also any difference in the value of the intermediate result (caused by different rounding behavior or greater intermediate accuracy) can affect the final result, which might have raised an exception had two independent instructions been used.

C++

The contraction of expressions is not explicitly discussed in the C++ Standard.

---

footnote 76

76) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. 973

Commentary

The difference between machine instructions that combine multiple operators taking floating-point operands and those taking integer operands is that in the former case the final result may be different. While an infinite number of combined processor instructions are possible, only a few combinations occur frequently in commercial applications. Also, while many combinations occur frequently, there are rarely any worthwhile performance advantages to be had from fusing them into a single instruction.

C90

Such instructions were available in processors that existed before the creation of the C90 Standard and there were implementations that made use of them. However, this license was not explicitly specified in the C90 Standard.

C++

The C++ Standard contains no such explicit license.

Other Languages

Although this implementation technique is not explicitly discussed in the Fortran Standard, some of its implementations make use of it.

Common Implementations

Some processors do have special integer instructions for handling graphics processing. However, these are not usually sufficiently general purpose (i.e., they are algorithm-specific) to be used for the combined evaluation of C operators. Combined arithmetic operations on integer data types do not appear in any processor known to your author.

While processors may have fused instructions available, implementations vary in their support for such instructions. The reason for this is that many of the processors providing such instructions are designed with

particular kinds of applications in mind and often have an irregular architecture. Even if fused instructions are generated by a translator, its pattern of register usage may be far from optimal because of the difficulties of mapping sequences of operators in a coherent way.

A study by Arnold and Corporaal[58] looked for frequently occurring sequences of operations in DSP applications. The results found that three operand arithmetic and load/store with prior address calculation occurred reasonably frequently (i.e., may be worth creating a single instruction to perform them).

---

974 As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

contraction undermine predictability

Commentary

This sentence applies to implementations. What exactly is a *well-defined* use? For instance, the intent of fused operations is increased performance. How much additional information is needed? Developers need information on the circumstances under which a translator will make use of such instructions and the impact of any surrounding subexpression evaluations (i.e., their potential impact on the generation of exceptions, NaNs, and underflow/overflow). Requiring developers to read listings of generated machine code probably does not count as clearly documented.

Algorithms created by floating-point experts take account of the rounding that will take place during the evaluation of an expression. If the expected rounding does not occur, the results may be less accurate than expected. This difference in rounding behavior need not be restricted to the contracted subexpression; one part of an expression may be contracted and another related part not contracted, leading to an imbalance in the expected values. (The decision on whether to contract can depend on the surrounding context, the same expression being contracted differently in different contexts, undermining predictability.)

Many developers do not have the mathematical sophistication needed to perform an error analysis of an algorithm containing one or more expressions that have been contracted (or not). The behavior of an implementation for particular algorithms is likely to be found, by developers, by measuring how it handles various test cases.

Other Languages

This is an issue that applies to all language implementations that make use of fused instructions to contract expressions.

Coding Guidelines

A clear, fully documented implementation provides no more rationale for allowing translators to contract expressions than a poorly documented one is a rationale for disallowing them. The guideline recommendations on contraction to apply whatever the state of the implementation's documentation.

Example

If the intermediate result of some operator is (with four additional bits of internal accuracy, shown using binary representation, with an underscore after position 53):

```
1.00000000000000000000001011111111111111111111111111111_0111
```

it would be rounded to:

```
1.00000000000000000000001011111111111111111111111111111
```

However, if the original result were not rounded, but immediately used as the operand of some form of fused add instruction with the other operand having the value (in practice the decimal point would be implicitly shifted right):

```
0.00000000000000000000000000000000000000000000000000000_0100
```

the result would be:

```
1.00000000000000000000011000000000000000000000000000
```

In the nonfused instruction case, where rounding occurred, the result of the add would be the value of the first operand.

6.5.1 Primary expressions

primary-expression syntax

975

```
primary-expression:
                identifier
                constant
                string-literal
                ( expression )
```

Commentary

A primary expression may be thought of as the basic unit from which a value can be read, or into which one can be stored. There is no simpler kind of expression (parentheses are a way of packaging up a complex expression).

C++

The C++ Standard (5.1p1) includes additional syntax that supports functionality not available in C.

Other Languages

Many languages treat a wider range of constructs as being primary expressions (because they do not define array selection, `[]`, structure and union member accesses, and function calls as operators).

Common Implementations

compound expression 1313

`gcc` supports what is calls *compound expressions* as a primary expression.

Semantics

identifier is primary expression if

976

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[77)]

Commentary

An identifier that has not been declared cannot be a *primary-expression*. So references to undeclared identifiers are a violation of syntax (a less through reading of the standard had led many people to believe that such usage was implicitly undefined behavior; the footnote was added in C99 to highlight this point) and must be diagnosed. Whether the identifier denotes a value or refers to an object depends on the operator, if any, of which it is an operand.

lvalue converted to value 725

C++

The C++ definition of identifier (5.1p7) includes support for functionality not available in C. The C++ Standard uses the term *identifier functions*, not the term *function designator*. It also defines such identifier functions as being lvalues (5.2.2p10) but only if their return type is a reference (a type not available in C).

Other Languages

Some languages implicitly declare identifiers that are not explicitly declared. For instance, Fortran implicitly declares identifiers whose names start with one of the letters `I` through `N` as integers, and all other identifiers as reals.

Common Implementations

Using current, high-volume, commodity technology, accessing an object's storage can be one of the slowest operations. Since 1986 CPU performance has increased by a factor of 1.55 per annum, while DRAM (one of the most common kinds of storage used) performance has only increased at 7% per annum[560] (see Figure 0.6). Many modern processors clock at a rate that is orders of magnitude faster than the random access memory chips they are interfaced to. This can result in a processor issuing a load instruction and having to wait 100 or more clock cycles for the value to be available for subsequent instructions to use. The following are a number of techniques are used to reduce this time penalty:

- Translators try to keep the values of frequently used objects in registers. 1369 register storage-class
- Hardware vendors add caches to their processors. In some cases there can be an on-chip cache and an off-chip cache, the former being smaller but faster (and more expensive) than the latter. 0 cache
- Processors can be designed to be capable of executing other instructions, which do not require the value being loaded, while the value is obtained from storage. This can involve either the processor itself deciding which instructions can be executed or its designers can expose the underlying operations and allow translators to generate code that can be executed while a value is loaded. For instance, the MIPS processor has a delay slot immediately after every load instruction; this can be filled with either a NOP instruction or one that performs an operation that does not access the register into which the value is being loaded. It is then up to translator implementors to find the sequence of instructions that minimizes execution time.[788]

Studies have found that a relatively small number of load instructions, so called *delinquent loads*, account for most of the cache misses (and therefore generate the majority of memory stalls). A study by Panait, Sasturkar, and Wong[1048] applied various heuristics to the assembler generated from the SPEC benchmarks to locate those 10% of load instructions that accounted for over 90% of all data cache misses. When basic bock profiling was used they were able to locate the 1.3% of loads responsible for 82% of all data cache misses.

Many high-performance processors now support a 64-bit data bus, while many programs continue to use 32-bit scalar types for the majority of operations. This represents a 50% utilization of resources. One optimization is to load (store is less common) two adjacent 32-bit quantities in one 64-bit operation. Opportunities for such optimizations are often seen within loops performing calculations on arrays. One study[13] was able to significantly increase the efficiency of memory accesses and improve performance based on this optimization.

Predicting the value of an object represented using a 32-bit word might be thought to have a 1 in 4×10^9 chance of being correct. However, studies have found that values held in objects can be remarkably predictable.[189,859]

Given that high-performance processors contain a cache to hold the value of recently accessed storage locations, the predictability of the value loaded by a particular instruction might not be thought to be of use. However, high-performance processors also pipeline the execution of instructions. The first stages of the pipeline perform instruction decoding and pass the components of the decoded instruction on to later stages, which eventually causes a request for the value at the specified location to be loaded. The proposed (no processors have been built— the existing results are all derived from the behavior of simulations of existing processors modified to use some form of value prediction tables) performance improvement comes from speculatively executing[466] other instructions based on a value looked up (immediately after an instruction is decoded and before it passes through other stages of the pipeline) in some form of *load value locality* table (indexed by the address of the load instruction). If the value eventually returned by the execution of the load instruction is the same as the one looked up, the results of the speculative execution are used; otherwise, the results are thrown away and there is no performance gain. The size of any performance gain depends on the accuracy of the value predictors used and a variety of algorithms have been proposed.[186,992] It has also been proposed that some value prediction decisions be made at translation time.[187]

value locality 0 cache 0 processor pipeline

Coding Guidelines

Some coding guidelines documents require that all identifiers be declared before use. This requirement arises from the C90 specification that an implicit declaration be provided for references to identifiers, which had not been declared, denoting function designators. Such an implicit declaration is not required in C99 and a conforming implementation will issue a diagnostic for all references to undeclared identifiers. This issue is discussed elsewhere.

operator 1000 ()

Usage

A study by Yang and Gupta[1489] found, for the SPEC95 programs, on average eight different values occupied 48% of all allocated storage locations throughout the execution of the programs. They called this behavior *frequent value locality*. The eight different values varied between programs and contained small values (zero was often the most frequently occurring value) and very large values (often program-specific addresses of objects and string literals).

Table 976.1: Dynamic percentage of load instructions from different *classes*. The *Class* column is a three-letter acronym: the first letter represents the region of storage (Stack, Heap, or Global), the second denotes the kind of reference (Array, Member, or Scalar), and the third indicates the type of the reference (Pointer or Nonpointer). For instance, *HFP* is a load of pointer-typed member from a heap-allocated object. There are two kinds of loads generated as a result of internal translator housekeeping: *RA* is a load of the return address from a function-call, and any register values saved to memory prior to the call also need to be reloaded when the call returns, *CS* callee-saved registers The figures were obtained by instrumenting the source prior to translation. As such they provide a count of loads that would be made by the abstract machine (apart from *RA* and *CS*). The number of loads performed by the machine code generated by translators is likely to be optimized (evaluation of constructs moved out of loops and register contents reused) and resulting in fewer loads. Whether these optimizations will change the distribution of loads in different classes is not known. Adapted from Burtscher, Diwan and Hauswirth.[187]

Class	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex	bzip	gzip	mcf	Mean
SSN	–	1.28	3.50	0.42	4.40	12.10	6.23	7.26	0.12	0.15	0.15	2.97
SAN	–	0.63	1.01	16.61	–	0.45	2.58	–	12.73	0.01	–	2.84
SMN	–	0.67	–	3.62	–	0.30	–	2.60	–	–	–	0.60
SSP	–	0.37	–	0.17	1.40	–	–	0.33	–	0.02	–	0.19
SAP	–	0.25	–	0.17	–	–	–	–	–	–	–	0.04
SMP	–	0.29	–	0.25	0.01	0.24	2.15	0.05	–	–	–	0.25
HSN	–	0.88	–	14.75	3.51	–	8.07	7.32	0.27	0.01	0.20	2.92
HAN	–	7.39	–	48.55	–	–	4.30	5.39	31.83	–	2.75	8.35
HMN	–	16.37	–	0.76	8.80	6.11	8.42	0.85	–	3.54	27.35	6.02
HSP	–	0.33	–	–	1.82	–	20.01	7.64	–	–	–	2.48
HAP	–	9.42	–	1.33	0.56	–	3.02	4.97	–	–	0.88	1.68
HMP	–	1.82	–	0.11	24.44	0.57	6.29	0.16	–	0.01	17.47	4.24
GSN	43.46	11.10	14.23	0.45	12.76	17.49	16.81	27.79	43.71	43.75	3.12	19.56
GAN	19.27	6.51	52.03	3.00	–	21.86	–	0.03	3.63	26.24	–	11.05
GMN	–	0.81	–	0.41	–	10.96	–	0.16	–	–	2.79	1.26
GSP	–	0.68	–	0.04	–	–	–	–	–	–	0.48	0.10
GAP	–	2.17	–	–	–	0.86	–	0.60	0.41	–	4.72	0.73
GMP	–	0.77	–	0.20	–	0.07	–	–	–	–	0.26	0.11
RA	7.65	5.16	3.68	0.91	8.84	4.58	4.11	4.60	0.76	2.52	7.29	4.17
CS	29.62	33.10	25.55	8.27	33.46	24.40	18.01	30.24	6.54	23.75	32.55	22.12

function leaf/non-leaf

A common program design methodology specifies that all the work should be done in the leaf functions (a *leaf function* is one that doesn't call any other functions). The nonleaf functions simply forms a hierarchy that calls the appropriate functions at the next level. In their study of the characteristics of C and C++ programs (using SPECint92 for C), Calder, Grunwald, and Zorn[192] made this leaf/nonleaf distinction when reporting their findings (see Table 976.2).

instruction 0 profile for different processors

The issue of dynamic instruction characteristics varying between processors and translators is discussed elsewhere. In the case of load instructions, Table 976.3 compares runtime percentages for two different processors.

Table 976.2: Occurrence of load instructions (as a percentage of all instructions executed on HP–was DEC– Alpha). The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* is for calls to nonleaf functions. Adapted from Calder, Grunwald, and Zorn.[192]

Program	Mean	Leaf	NonLeaf	Program	Mean	Leaf	Non-Leaf
burg	21.7	12.9	26.7	eqntott	12.8	11.8	20.2
ditroff	30.3	18.6	32.9	espresso	21.6	20.1	22.9
tex	30.7	19.6	31.3	gcc	23.9	16.7	24.6
xfig	23.5	15.6	25.8	li	28.1	44.1	26.3
xtex	23.2	16.1	28.2	sc	21.2	15.3	22.8
compress	26.4	0.1	26.5	Mean	23.9	17.3	26.2

Table 976.3: Comparison of percentage of load instructions executed on Alpha and MIPS. Adapted from Calder, Grunwald, and Zorn.[192]

Program	MIPS	Alpha	Program	MIPS	Alpha
compress	17.3	26.4	li	21.8	28.1
eqntott	14.6	12.8	sc	19.2	21.2
espresso	17.9	21.6	Program mean	18.2	22.3
gcc	18.7	23.9			

977 A constant is a primary expression.

Commentary

A constant is a single token. A constant expression is a sequence of one or more tokens.

1322 constant expression syntax

Common Implementations

The numeric values of most constants that occur in source code tend to be small. Processor designers make use of this fact by creating instructions that contain a constant value within their encoding. In the case of RISC processors, these instructions are usually limited to loading constant values into a register (the constant zero occurs so often that many of them dedicate a, read-only, register to holding this value). Many CISC processors having instructions to perform arithmetic and logical operations, the constant value being treated as one of the operands. For instance, the Motorola 68000[968] had an optimized add instruction (`ADDQ`, add quick) that included three bits representing values between 1 and 8, in addition to the longer instructions containing 8-, 16-, and 32-bit constant values.

Coding Guidelines

Guidelines often need to distinguish between constants that are visible in the source code and those that are introduced through macro replacement. The reason for this difference in status is caused by how developers interact with source code; they look at the source code prior to translation phase 1, not as it appears after preprocessing. The issues involved in giving symbolic names to constants are discussed elsewhere.

macro replacement
822 symbolic name

Usage

Usage information on the distribution of all constant values occurring in the source is given elsewhere.

825 integer constant syntax

978 Its type depends on its form and value, as detailed in 6.4.4.

Commentary

Syntactically all constants have an arithmetic type (the null pointer constant either has the form of an octal constant, or a cast of such a constant, which is not a primary expression).

822 constant syntax

979 A string literal is a primary expression.

Commentary

The only context in which a string literal can occur in source code is as a primary expression.

Usage

string literal 895 syntax

Usage information on string literals is given elsewhere.

It is an lvalue with type as detailed in 6.4.5. 980

Commentary

lvalue 721 string literal 904 type

It is an lvalue because it has an object type and its type depends on the lexical form of the string literal.

---

parenthesized expression

A parenthesized expression is a primary expression. 981

Commentary

Parentheses can be thought of as encapsulating the expression within them.

expression 189 evaluation abstract machine base document 1

Implementations are required to honor the operator/operand pairings of an expression implied by the presence of parentheses. The base document differs from the standard in allowing implementations to rearrange expressions, even in the presence of parentheses.

Common Implementations

An optimizer may want to reorder the evaluation of operands in an expression to improve the performance or size of the generated code. For instance, in:

```
1  extern int i, j, k;
2
3  void f(void)
4  {
5  int x = i + k,
6      y = 21 + i + j + k;
7  /* ... */
8  }
```

it may be possible to improve the generated machine code by rewriting the subexpression `21 + i + j + k` as `i + k + j + 21`. Perhaps the result of the evaluation of `i + k` is available in a register, or is more quickly obtained via the object `x`.

However, such expression rewriting by a translator may not preserve the intended behavior of the expression evaluation. The expression may have been intentionally written this way because the developer knew that the evaluation order specified in the standard would guarantee that the intermediate and final results were always representable (`i + k` may be a large negative value, with `j` sometimes having a value that would cause this sum to overflow).

In the above case, rewriting as `((21 + i) + j) + k` would stop most optimizers from performing any reordering of the evaluation. However, if an optimizer can deduce that reordering the evaluation through parentheses would not affect the final result, it can invoke the as-if rule (on processors where signed integer arithmetic wraps and does not signal an overflow, reordering the evaluation of the expression would not cause a change of behavior). The only visible change in external behavior might be a change in program performance, or a smaller program image.

as-if rule 122

For floating-point types wrapping behavior cannot come to an optimizer's rescue (by enabling overflows to be ignored). However, some implementations may chose to consider overflow as a rare case, preferring the performance advantages in the common cases. Overflow is not the only issue that needs to be considered when operands have a floating-point type, as the following example shows:

```
1  extern double i, j, k;
2
3  void f(void)
```

```
{
double x = i + k,
       y = i + j + k;
/* ... */
}
```

assuming the objects have the following values:

```
i = 1.0E20
j = -1.0E20
k = 6.0
```

then the value that is expected to be assigned to `y` is `6.0`. Rewriting the expression as `i + k + j` would result in the value `0.0` being assigned (because of limited accuracy, adding `6.0` to `1.0E20` gives the result `1.0E20`).

Coding Guidelines

There is a guideline recommendation specifying that expressions shall be parenthesized.

943.1 expression shall be parenthesized

Use of parentheses makes the intentions of the developer clear to all. Objections raised, by developers, against the use of unnecessary parentheses are often loud and numerous. Many developers consider that use of parentheses needs to be justified on a case-by-case basis. Others take the view that there are no excuses for not knowing the precedence of all C operators and that all developers should learn them by heart. These coding guidelines accept that many developers do not know the precedence of all C operators and that exhortations to learn them will have little practical effect. Parenthesizing all binary (and some unary) operators and their associated operands is considered to be the solution (to the problem of developers' incorrect deduction of the grouping of operands within an expression). In some instances a case can be made for not using parentheses, which are discussed in the Syntax sections of the relevant operators.

943 operator precedence

As the discussion in Common Implementations showed, the use of parentheses can sometimes reduce the opportunities available to an optimizer to generate more efficient machine code. These coding guidelines consider this to be a minor consideration in the vast majority of cases, and it is not given any weight in the formulation of any guideline recommendations.

Some coding guideline documents recommend against redundant parentheses; for instance, in `((x))` the second set of parentheses serves no purpose. Occurrences of redundant parentheses are rare and there is no evidence that they have any significant impact on source code comprehension. These coding guidelines make no such recommendation.

Usage

Usage information on parentheses usage is given elsewhere.

281 parenthesized expression nesting levels

---

982 Its type and value are identical to those of the unparenthesized expression.

Commentary

But, for the use of expression rewriting by an optimizer, the generated machine code will also be identical.

Common Implementations

Some early implementations considered that parentheses 'hid' their contents from subsequent operators. This created a difference in behavior between `sizeof("0123456")` and `sizeof(("0123456"))`— one returning the **`sizeof`** of an array operand, the other the size of a pointer operand. The C Standard makes no such distinction.

729 array converted to pointer

---

983 It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

Commentary

This means that `(a) = (f)(x)` and `a = f(x)` are equivalent constructs.

Other Languages

In many languages a function call is a primary expression, so it is not possible to parenthesize a function designator.

Forward references: declarations (6.7). 984

6.5.2 Postfix operators

postfix-expression syntax

985

```
postfix-expression:
                primary-expression
                postfix-expression [ expression ]
                postfix-expression ( argument-expression-list_opt )
                postfix-expression .  identifier
                postfix-expression -> identifier
                postfix-expression ++
                postfix-expression --
                ( type-name ) { initializer-list }
                ( type-name ) { initializer-list , }
argument-expression-list:
                assignment-expression
                argument-expression-list , assignment-expression
```

Commentary

The token pairs `[ ]` and `( )` are not commonly thought of as being operators.

C90

Support for the forms (compound literals):

```
( type-name ) { initializer-list }
( type-name ) { initializer-list , }
```

is new in C99.

C++

Support for the forms (compound literals):

```
( type-name ) { initializer-list }
( type-name ) { initializer-list , }
```

is new in C99 and is not specified in the C++ Standard.

Other Languages

Many languages do not treat the array subscript (`[]`), structure and union member accesses (`.` and `->`), or function calls (`()`) as operators. They are often included as part of the syntax (as punctuators) for primary expressions. The syntax used in C for these operators is identical or very similar to that often used by other languages containing the same construct.

Many languages support the use of comma-separated expressions within an array index expression. Each expression is used to indicate the element of a different dimension— for instance, the C form `a[i][j]` can be written as `a[i, j]`.

Some languages use parentheses, `()`, to indicate an array subscript. The rationale given for using parentheses in Ada[619] is based on the principle of *uniform referents*— a change in the method (i.e., a function call or array index) of evaluating an operand does not require a change of syntax.

Cobol uses the keyword **OF** to indicate member selection. Fortran 95 uses the **%** symbol to represent the `->` operator.

Perl allows the parentheses around the arguments in a function call to be omitted if there is a declaration of that function visible.

Common Implementations

The question of whether using the postfix or prefix form of the `++` and `--` operators results in the more efficient machine code crops up regularly in developer discussions. The answer can depend on the processor instruction set, the translator being used, and the context in which the expression occurs. It is outside the scope of this book to give minor efficiency advice, or to list all the permutations of possible code sequences that could be generated for specific operators.

Coding Guidelines

There is one set of cases where developers sometime confuse the order in which prefix operators and *unary-operators* are applied to their operand. The expression `*p++` is sometimes assumed to be equivalent to `(*p)++` rather than `*(p++)`. A similar assumption can also be seen for the postfix operator `--`. The guideline recommendation dealing with the use of parenthesis is applicable here.

1080 unary-expression syntax

Dev 943.1 A postfix-expression denoting an array subscript, function call, member access, or compound literal need not be parenthesized.

Dev 943.1 Provided the result of a postfix-expression, denoting a postfix increment or postfix decrement operation, is not operated on by a unary operator it need not be parenthesized.

Example

```
1  struct s {
2           struct s *x;
3           };
4  struct s *a,
5           *x;
6
7  void f(void)
8  {
9  a>x;
10 a->x;
11 a-->x;
12 a--->x;
13 }
```

Table 985.1: Occurrence of postfix operators having particular operand types (as a percentage of all occurrences of each operator, with **[** denoting array subscripting). Based on the translated form of this book's benchmark programs.

Operator	Type	%	Operator	Type	%
v++	**int**	54.0	**[**	**unsigned char**	5.1
v--	**int**	52.5	**[**	other-types	4.7
[	*****	38.0	**[**	**int**	4.1
v++	*****	25.7	**v++**	**unsigned long**	3.1
v--	**long**	15.9	**v--**	**unsigned short**	2.7
[	**struct**	14.5	**v--**	**unsigned char**	2.6
v++	**unsigned int**	13.3	**[**	**const char**	2.4
[	**float**	12.0	**[**	**unsigned long**	1.2
v--	**unsigned int**	11.5	**v++**	**long**	1.1
[	**union**	10.2	**[**	**unsigned int**	1.1
v--	*****	7.1	**v++**	**unsigned short**	1.0
[	**char**	6.8	**v++**	**unsigned char**	1.0
v--	**unsigned long**	6.1	**v--**	**short**	1.0

Table 985.2: Common token pairs involving `.`, `->`, `++`, or `--` (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier `->`	9.8	97.5	`v++ )`	41.4	1.4
identifier `v++`	0.9	96.9	`v++ ;`	39.9	1.4
identifier `v--`	0.1	96.1	`v++ ]`	4.6	1.3
identifier `.`	3.6	83.8	`v++ =`	7.6	0.7
`] .`	20.3	15.4	`v-- ;`	58.4	0.3
`->` identifier	100.0	10.1	`v-- )`	29.1	0.1
`.` identifier	100.0	4.2			

footnote 77

77) Thus, an undeclared identifier is a violation of the syntax. 986

Commentary

This fact was not explicitly pointed out in the C90 Standard, which led some people to believe that the behavior was undefined. The response to DR #163 specified the order in which requirements, given in the standard, needed to be read (to deduce the intended behavior).

C++

The C++ Standard does not explicitly point out this consequence.

Other Languages

Some languages regard a reference to an undeclared identifier as a violation of the language semantics that is required to be diagnosed by an implementation.

Common Implementations

Most implementations treat undeclared identifiers as primary expressions. It is during the subsequent semantic processing where the diagnostic associated with this violation is generated. The diagnostic often points out that the identifier has not been declared rather than saying anything about syntax.

6.5.2.1 Array subscripting

Constraints

subscripting

One of the expressions shall have type "pointer to object *type*", the other expression shall have integer type, and the result has type "*type*". 987

Commentary

Surprisingly there is no requirement that the pointer type be restricted to occurring as the left operand (because such a requirement invariably exists in other computer languages).

Other Languages

Most languages require that the left operand have an array type. The implicit conversion of arrays to pointers is unique to C (and C++). A few languages (e.g., Awk, Perl, and Snobol 4) support the use of strings as indexes into arrays (they are called *table*s in Snobol 4).

Coding Guidelines

Although the C Standard permits the operands to occur in any order, the pointer type nearly always appears (in source) as the left operand. Developers have not gotten into the habit of using any other operand ordering. Introductory books on C teach this operand order and many don't even mention that another order is permitted. There is nothing to be gained by specifying a particular operand order in a coding guideline; the alternative is very rarely seen.

Example

The following all conform to this requirement:

```
1   extern int arr[10];
2   extern int glob;
3
4   void f(void)
5   {
6   (glob++)[arr] = 9;
7   arr[glob-1]=10;
8   (arr+2)[3]=4["abcdefg"];
9   "abc"[2]='z';           /* Ok, undefined behavior. */
10  }
```

Semantics

988 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object.

Commentary

Many developers do not think of square brackets, `[]`, as being an operator. A single token would be sufficient to indicate an array subscript. However, existing practice in other languages and the advantages of the bracketing effect of using two tokens (it removes the need to use parentheses when the subscript expression is more complex than a *unary-expression*) were more important considerations.

Subscripted arrays are not as commonly seen in C source code as they are in programs written in other languages. Using pointers where arrays would be used in other languages, seems like a more natural fit in C. Among the reasons for this might be the support for pointer arithmetic available in C but not many other languages, and the automatic conversion of arrays to pointers to their first element.

1165 additive operators pointer to object

Other Languages

Many languages allow all of the subscripts in a multidimensional array access to appear within a single pair of square brackets (or parentheses for some languages), with each subscript being separated by a comma (rather like the arguments in a function call). In some languages subscripting an array is implied by the appearance of an expression to the right of an object having an array type.

1577 footnote 121

Common Implementations

Most processor instruction sets support one or more forms of addressing that is designed to handle the accessing of storage via array subscripting. In practice the *pointer-to object type* is often a value that is known at translation time (e.g., a file scope array object whose address is decided at link time). In this case the subscript value can be loaded into a register and a *register with displacement* addressing mode used (an almost universally available addressing mode).

Values contained within arrays are often accessed sequentially, one at a time. A common C practice is to assign the base address of the array to a pointer, access each element through that pointer, incrementing the pointer to move onto the next element.

```
1   #define NUM_ELEMS (20)
2
3   extern int a[NUM_ELEMS];
4
5   void f(void)
6   {
7   int *p = a;
8
9   while (p != a+NUM_ELEMS)
10     /*
```

```
11       * Do something involving:
12       *
13       *         *p
14       *
15       * p will need to be incremented before the loop goes around again.
16       */
17      ;
18
19   for (int a_index=0; a_index < NUM_ELEMS; a_index++)
20      /*
21       * Do something involving:
22       *
23       *         a[a_index]
24       */
25      ;
26   }
```

data dependency

Depending on the processor instruction set and the surrounding source code context, use of a pointer may result in more or less efficient machine code than an array access. However, looking at the code in a wider perspective, the use of pointers rather than arrays makes some optimizations significantly more difficult to implement. The problem is one of possible data dependencies— an optimizer needs to know what they are; and the analysis is significantly more difficult to perform when pointers rather than arrays are involved.

Scientific and engineering programs often spend a large amount of time within loops reading and writing array elements. Such programs tend to loop through elements of different arrays, performing some calculation involving each of them. When a processor can execute more than one instruction at the same time, finding the most efficient ordering is technically very difficult. The problem is knowing when the value read from an array element is going to be affected by the writing of a value to the same array. Knowing that there is no dependency between two accesses allows an optimizer to order them as it sees fit. (If there is a dependency, the operations must occur in the order as written.) The sequence of statements within a loop may also have been unrolled, exposing dependencies between accesses in what were different iterations. The patterns of array reference usage have been studied in an attempt to generate faster machine code. The three types of array reference patterns are:

loop unrolling 1774

Flow-dependent

```
1   for (index = 1; index < 10; index++)
2      {
3      A[index-1] = B[index];
4      C[index] = A[index];
5      }
```

Anti-dependent

```
1   for (index = 1; index < 10; index++)
2      {
3      B[index] = A[index-1];
4      A[index] = C[index];
5      }
```

Output-dependent

```
1   for (index = 1; index < 10; index++)
2      {
3      A[index-1] = B[index];
4      A[index] = C[index];
5      }
```

It is not necessary to be targeting a special-purpose parallel processor to want to try to optimize these loops. Modern processors have multiple execution integer and floating-point arithmetic units.[6, 350, 627, 1342] Keeping all units busy can result in significant performance improvements.

For the flow-dependent case: Unrolling the loop once, we see the order of execution for the assignment statements is:

Time step	Operation
t=1	A[0] = B[1]
t=2	C[1] = A[1]
t=3	A[1] = B[2]
t=4	C[2] = A[2]

Attempting to perform these two iterations in parallel, we get:

Time step	Thread 1	Thread 2
t=1	A[0] = B[1]	A[1] = B[2]
t=2	C[1] = A[1]	C[2] = A[2]

There is an assignment to `A[1]` in execution thread 2 at time *t=1* before that value is used in execution thread 1 at time *t=2*. Executing this code in parallel would cause the array element value to be given a new value before its previous value had been used. A modification to the loop, known as *preloading*, removes this flow dependency:

```
1  for (index = 1; index < 10; index++)
2     {
3     T = A[index];
4     A[index-1] = B[index];
5     C[index] = T;
6     }
```

For the anti-dependency case: Unrolling the loop once, we see the order of execution for the assignment statements is:

Time step	Operation
t=1	B[1] = A[0]
t=2	A[1] = C[1]
t=3	B[2] = A[1]
t=4	A[2] = C[2]

Attempting to perform these two iterations in parallel, we get:

Time step	Thread 1	Thread 2
t=1	B[1] = A[0]	B[2] = A[1]
t=2	A[1] = C[1]	A[2] = C[2]

Here the value assigned to `B[2]` in execution thread 2 at time *t=1* is the old value of `A[1]`, before it is updated in execution thread 1 at time *t=2*. Executing this code in parallel would cause the array element value to be given a value that is incorrect, the correct one not yet having been calculated. Reordering the sequence of assignments removes this anti-dependency:

```
1  for (index = 1; index < 10; index++)
2     {
3     A[index] = C[index];
4     B[index] = A[index-1];
5     }
```

For the output dependent case: Unrolling the loop once, we see the order of execution for the assignment statements is:

Time step	Operation
t=1	A[0] = B[1]
t=2	A[1] = C[1]
t=3	A[1] = B[2]
t=4	A[2] = C[2]

Attempting to perform these two iterations in parallel, we get:

Time step	Thread 1	Thread 2
t=1	A[0] = B[1]	A[1] = B[2]
t=2	A[1] = C[1]	A[2] = C[2]

Here the correct assignment to `A[1]` in execution thread 2 at time *t=1* is overwritten by an assignment in execution thread 2 at time *t=2*. Executing the code in parallel causes the final value of the array element to be incorrect. Reordering the sequence of assignments removes this output dependency:

```
1  for (index = 1; index < 10; index++)
2     {
3     A[index] = C[index];
4     A[index-1] = B[index];
5     }
```

It is not usually intuitively obvious if a dependency exists between elements of an array in different iterations of a loop. A number of tests based on the mathematics of number theory are known; for instance, in:

```
1  for (index = 5; index < 10; index++)
2     {
3     A[2*index - 1] = C[index];
4     B[index] = A[4*index - 7];
5     }
```

if a dependency exists, there must be values of `x` and `y` such that the element accessed in statement `S(x)` is the same as that in statement `S(y)`. For this to occur the relation $2x - 1 = 4y - 7$ must hold. This occurs when x and y equal 3, which is not within the bounds of the loop, so there is no dependency in this case.

Equations such as these, known as *Diophantine* equations, are usually written with the variable terms on the left and the constant value on the right (e.g., $4y - 2x = 6$). It is known that a solution to such an equation, in integer values, exists if and only if the greatest common divisor of the constants on the left hand side divides the constant on the right-hand side (which it does in this case). This test, known as the *GCD* test, is very simple to apply and will determine whether there is a dependency between accesses to an array. However, the problem with the GCD test is that it does not fully take advantage of the known information (the bounds of the loop). The *Banerjee* test[89] is a much more sophisticated test that is often performed (others have also been proposed).

Even with these sophisticated tests, it is still only possible to deduce whether dependencies exist for a fraction of the cases commonly seen in industrial applications.[1224] Allen and Kennedy[17] discuss data dependency in detail, specifically algorithms for optimizing the ordering of array accesses within loops (Fortran-based). For calculations involving sparse matrices, it is sometimes possible to improve performance and reduce storage overhead by mapping the representation of the nonzero array elements to another kind of data structure.[123]

Example

```
1  extern int a[3][4];
2  extern int (*p1)[];
3  extern int *p2;
4  extern int p3;
```

```
5
6   void f(void)
7   {
8   p1=a;    /* An array of array of int becomes a pointer to an array of int. */
9
10  p2=a[1]; /* Indexing exposes the array of int, which is converted to pointer to int. */
11
12  p3=a[1][2]; /* Indexing again reveals an int. */
13
14  if (a[1][2] != p2[2]) /* p2 points at the start of a[1] */
15     printf("Something wrong here\n");
16  }
```

---

989 The definition of the subscript operator **[]** is that **E1[E2]** is identical to **(*((E1)+(E2)))**.

array subscript identical to

Commentary

This explains the symmetry between the operands of the **[]** operator. In their equivalent form, operand order is irrelevant. This equivalence also highlights the point that C requires as much checking on the bounds of array subscripts by implementations as it requires on pointer differencing.

Other Languages

This equivalence relationship is unique to C (and C++).

Common Implementations

Some translators rewrite array accesses into this form in their internal representation. It reduces the number of different cases that need to be considered during code generation.

The degree to which the use of subscripted arrays or pointers affects the quality of generated machine code will depend on the sophistication of the analysis performed by a translator. Simpler translators usually generate higher-quality machine code when pointers are used. More sophisticated optimizers can make use of the information provided by a subscripted array (the name of the object being accessed) to generate higher-quality code. A study by Franke and O'Boyle[441] obtained up to 14% improvement in execution time (for several DSPstone benchmarks[1395] using the same translator) by source-to-source transformation, converting pointer accesses to explicit array accesses prior to processing by the compiler proper.

Coding Guidelines

Some guidelines documents recommend against the use of pointer arithmetic. Such a recommendation overlooks the equivalence between array subscripting and pointer arithmetic. While, it could be argued that this equivalence relationship is purely a specification detail in the C Standard, this would not correspond to how many developers think about array subscripting. This stated equivalence shows the extent to which the use of pointers, rather than arrays, are embedded within the C language. Coding guidelines that simply recommend against the use of pointer arithmetic are generally unworkable in practice. If developers are using C, guidelines need to work within the framework of that language.

---

990 Because of the conversion rules that apply to the binary + operator, if **E1** is an array object (equivalently, a pointer to the initial element of an array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from zero).

Commentary

It is sometimes claimed that having arrays zero based results in more efficient machine code being generated. Basing the array at zero can result in a small efficiency improvement for those cases where the address of the array is not known at translation (or link) time (machine code does not need to be generated to subtract one from the index expression). However, this analysis ignores the impact of the algorithm on the index expression. Any algorithm which naturally uses an array indexed from one, requires the developer to either

adjust the index expression to make it zero-based or to ignore the zero'th element (the array will contain an unused element). Whether more algorithms are naturally one-based rather than zero-based is not known.

Given the general interchangeability of arrays and allocated storage, the only practical option is for arrays to be zero-based. Arrays based at one would require that all allocated storage also have an implied base of one. In a language supporting pointer arithmetic, nonzero based arrays and pointers would significantly complicate the generated machine code and the developer's conception of where pointers actually pointed.

```
1  void f(void)
2  {
3  char a[10];
4  char *p = a;
5
6  a[1] = 'q';
7  p[1] = 'q';
8  *p = 'q';
9  p++;
10 *p = 'q';
11 }
```

Other Languages

Some languages base their array at one (e.g., Fortran). Languages in the Algol family allow the developer to specify both the lower and upper bounds of an array. For instance, in Pascal the definition `a[4..9] :Integer;` would define an array of integers that was to be subscripted with values between four and nine, inclusive (the underlying implementation would be zero-based, the translator generating code to subtract four from the index expression).

Coding Guidelines

Off-by-one coding errors may be the most common problems associated with array subscripting. This problem seems to be generic to all programming languages, independent of whether arrays start at zero, one, or a user defined value.

Successive subscript operators designate an element of a multidimensional array object. 991

Commentary

footnote 121 1577 Just like using the structure member selection operator (`.`) to access successive members of a nested structure, the `[]` operator can be used to select successive elements of a multidimensional array. A two-dimensional array can be thought of as adjacent slices of a single dimension array. As suggested by the mathematical terminology *dimension*, an array subscript can be thought of as the coordinates of the element being accessed in the defined array object.

For an array of arrays the word *object* denotes the specific object determined directly by the pointer's type and value, not other objects related to that one by contiguity. Therefore, if an array index exceeds the declared bounds, the behavior is undefined.

```
1  void DR_017_Q16(void)
2  {
3  int a[4][5];
4
5  a[1][7] = 0; /* Undefined behavior. */
6  }
```

Other Languages

Many languages require that all subscripts be given inside one pair of `[]` brackets. Such languages do not usually treat `[]` as an operator. A few languages provide operators that enable subsets of an array's elements to be selected. For instance, the Fortran 95 array selection `B(1:4, 6:8:2, 3)` specifies the set of elements whose first subscript varies between 1 and 4, the second subscript is either 6 or 8 (the third value, 2, in the index is a stride), and the third subscript is 3.

Common Implementations

Accesses to a multidimensional array requires that the translator calculate the address of the indexed element within the array object. In:

```
1  extern int a[3][5][7];
2  extern int j,
3             k,
4             l;
5
6  void f(void)
7  {
8  int i = a[j][k][l];
9  }
```

the array reference is equivalent to `(*(a+((((j*3)+k)*5)+l)))`. The value of this expression will also need to be multiplied by `sizeof(int)` to calculate the start of the element address.

When accesses to a multidimensional array occur within a nested loop, the evaluation of the indexing expression can often be optimized. For instance, in the following:

```
1  for (j=0; j<MAX_X; j++)
2     for (k=0; k<MAX_Y; k++)
3        for (l=0; l<MAX_Z; l++)
4           a[j][k][l]=formula;
```

parts of the array index calculation are the same for each iteration of some of the loops (e.g., `j*3` has a constant value during the iteration of the two nested loops, as does `((j*3)+k)*5` during the iteration of the innermost loop). It may be worthwhile to keep one or both of these values in a register, or save either of them to a temporary storage location (which at worst is likely to be faster than loading the loop control variable and performing a multiply).

1774 loop control variable

Coding Guidelines

Accessing the elements of a multidimensional array requires additional instructions to scale the index. A technique sometimes used by developers to reduce this overhead, when accessing all the elements (e.g., to assign an initial value), is to treat the array as if it had a single dimension.

```
1  #define NUM_1 10
2  #define NUM_2 20
3
4  int a[NUM_1][NUM_2];
5
6  void init(void)
7  {
8  int *q = (int *)a;
9
10 /*
11  * Some developers prefer using pointer arithmetic here.
12  */
13 for (int index=0; index < NUM_1*NUM_2; index++)
14    q[index]=0;
15 }
```

Use of this technique can be an indicator of an unnecessary interest in efficiency by developers. The guideline recommendation dealing with use of representation information is applicable here.

569.1 representation information using

Table 991.1: Occurrence of object declarations having an array type with the given number of dimensions (as a percentage of all array types in the given scope; with local scope separated into parameters and everything else). Based on the translated form of this book's benchmark programs.

Dimensions	Parameters	Local	non-parameter
Scope	File Scope		
1	100.0	97.9	91.9
2	0.0	2.0	7.5
3	0.0	0.1	0.6

array
n-dimensional
reference

If **E** is an *n*-dimensional array (*n*≥*2*) with dimensions $i \times j \times \cdots \times k$ then **E** (used as other than an lvalue) is converted to a pointer to an (*n*-1)-dimensional array with dimensions $j \times \cdots \times k$ 992

Commentary

Because this conversion does not occur when E is used as an lvalue, it is not possible to simultaneously assign multiple array elements. For instance, in:

```
int a[10][20];
int b[20];

/*
 * The left operand has type array of int.
 * The right operand has type pointer to int.
 */
a[4]=b;
```

there is a type mismatch. It is not possible to assign all of b's elements in one assignment statement.

C++

Clause 8.3.4p7 uses the term *rank* to describe $i \times j \times \cdots \times k$, not *dimensions*.

Other Languages

In most languages arrays are never converted to a pointer to their first element. Like C++, some other languages use the term *rank*.

Coding Guidelines

The equivalence relationship between array subscripting and pointer accesses sometimes leads to developer confusion over when arrays and pointers types can be interchanged. A multidimensional array is not the same as an array of pointers or a pointer-to pointer. Both of these are common, incorrect assumptions made by inexperienced developers. In:

```
int a[10][20];
int (*q)[];
int **p;

void f(void)
{
q=a;        /* a is a pointer to int, the first element of the first row. */
p=a;        /* Incompatible types, a is not a pointer to pointer to int.  */
p=(int *)a; /* Translator is not required to issue a diagnostic here.     */
}
```

Mixing these different types will result in diagnostic messages being generated. While developers have been known to use explicit casts to stop these diagnostics from appearing, it is likely that the resulting programs will fail to work correctly and a guideline recommendation addressing this usage is not considered worthwhile.

Example

```
1  int a[10][20][30];
2
3  void f(void)
4  {
5  int (*p1)[20][30] = a;
6  int (*p2)[30] = a[0];
7  /* ... */
8  }
```

---

993 If the unary * operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to (*n - 1*)-dimensional array, which itself is converted into a pointer if used as other than an lvalue.

Commentary

This process can continue for as many dimensions as there are in the array type. The last dereference yields the value of the pointed-to element (if used as other than an lvalue). The unary * operator has the same effect as subscripting the array with a zero index, `[0]`. This fact can be deduced from the definition of the subscript operator.

989 array subscript identical to

C++

*If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to ($n-1$)-dimensional array, which itself is immediately converted into a pointer.* 8.3.4p7

While the C++ Standard does not require the result to be used "as other than an lvalue" for it to be converted to a pointer. This difference does not result in any differences for the constructs available in C.

Coding Guidelines

While, technically, the forms * and `[0]` are interchangeable in many contexts, developers rarely think of them as such. For instance, many developers do not seem to separate out the components of an array definition as being both an allocation of storage and a creation of a reference to it (the array name). Are there any costs or benefits of only using the array subscript operator, **[]**, for operands having an array type and only the unary * operator for operands having a pointer type? Are there any costs or benefits of using one of these operators for both cases? The following are some of the issues:

- The notation `*a` is visually more compact than `a[0]`, while `a[index]` is shorter than `*(a+index)`.
- The unary * occurs much more frequently in C source code than the array subscript operator. It is possible that developers will be more practiced in the use of this form.
- In many contexts arrays are implicitly converted to pointers and arguments are always passed by value (not address). Less developer effort is needed to structure source code that uses pointers rather than arrays (which require decisions to be made on the number of elements to be specified in their declarations).

Example

Taking the example from the previous sentence, we have:

```
1  int a[10][20][30];
2
3  void f(void)
```

```
4  {
5  int (*p1)[30] = *a;
6  int (*p2) = *a[0];
7  /* ... */
8  }
```

array
row-major storage order

It follows from this that arrays are stored in row-major order (last subscript varies fastest). 994

Commentary

That is, the following equalities hold: `&a[i][j+1] == (&a[i][j]+1)` and `&a[i+1][j] == &a[i][j]+(sizeof(a[i][j]))`. Figure 994.1 illustrates the difference between row and column major storage layouts.

Figure 994.2 illustrates the difference in data structures used to represent the following two initialized object definitions in storage:

```
1  char day_arr[][10] = {
2                        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
3                        "Saturday", "Sunday"
4                        };
5  char *day_ptr[] =  {
6                     "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
7                     "Saturday", "Sunday"
8                     };
```

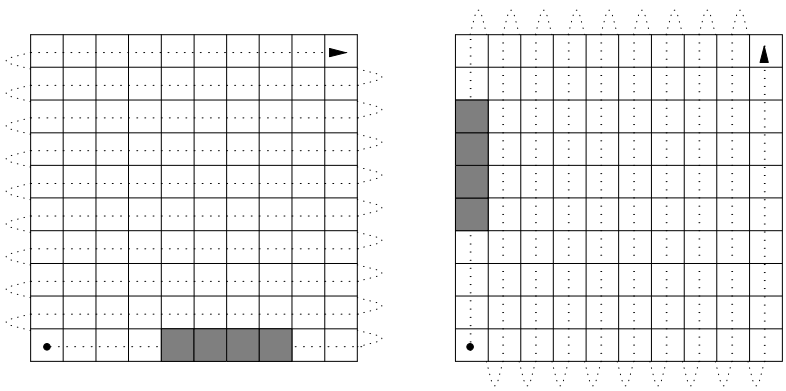


Figure 994.1: Row (left) and column (right) major order. The dotted line indicates successively increasing addresses for the two kinds of storage layouts, with the gray boxes denoting the same sequence of index values.

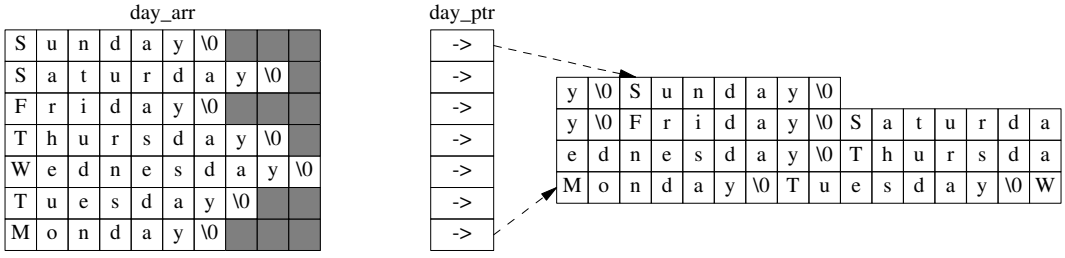


Figure 994.2: Difference in storage layout between an array of array of characters (left) and array of pointer to characters (right; not all pointers shown and the relative storage locations of the strings is only one of many that are possible).

The order in which array elements are organized can have significant performance implications. Processors use caches because their designers are aware that many data accesses exhibit spatial locality. If array elements are accessed in the order they are held in storage, maximum advantage can be taken of any available processor cache. The conditions under which the order of array element access might have an effect on program performance are discussed in the Common implementation section that follows. 0 cache

Other Languages

Some languages store arrays in column-major order (the first subscript varies the fastest). The most well-known being Fortran. One of the first things that developers of scientific and engineering applications, using Fortran, are told about is the importance of nesting loops to maintain spatial locality of reference.

Common Implementations

A processor's storage hierarchy can include a paged memory management system and often a cache. The performance of both of these can be affected by the order in which array elements are accessed. 0 cache

In a paged memory management system the amount of storage available for use by programs is usually greater than physically available. Memory is divided up into pages, usually 4 K or 8 K chunks, the less frequently used pages being swapped out to large, lower-cost storage (most often a hard disk). A large array can occupy many pages. Accessing noncontiguous array elements can require different pages needing to be in memory for each access. This can significantly impact performance because a swapped-out page consumes a lot of resources being swapped back into memory. Accessing elements in a contiguous order can significantly reduce the amount of paging activity.

A processor cache is usually structured to hold lines of data from memory. Each cache line holds a fixed number of bytes, usually some small power of 2 (but invariably larger than any scalar type). A load from an address will result in a complete cache line being fetched if the data at that address is not already in the cache. Accesses to sequential locations in storage can often be satisfied from data already in the cache.

```
1  #define INNER 1000
2  #define OUTER 2000
3
4  extern int a[INNER][OUTER];
5
6  void f(void)
7  {
8  /*
9   * This loop could execute more quickly than the one below
10  */
11 for (int i=0; i < INNER; i++)
12    {
13    /*
14     * Each access will be adjacent, in storage, to the previous one.
15     */
16    for (int j = 0; j < OUTER; j++)
17       a[i][j]+=1;
18    }
19
20 for (int j=0; j < OUTER; j++)
21    {
22    /*
23     * Each access will be disjoint, in storage, to the previous one.
24     */
25    for (int i = 0; i < INNER; i++)
26       a[i][j]+=1;
27    }
28 }
```

The preceding example illustrates spatial locality. A program's use of arrays can also show temporal locality. In the following example (function g), each element of the array c is accessed in the inner loop. If the number

of bytes in the array c is greater than the total size of the cache, then on the next iteration of the outer loop the values from the first elements will not be in the cache. Loading these values will cause other elements' values of c to be flushed from the cache.

A performance-driven solution is to introduce a nested loop. This loop iterates over a smaller range of values, sufficiently small that all the elements fit within the cache. This transform is known as *strip-mine and interchange* and is one of a family of so-called *cache blocking* transforms. Function h shows such a rewritten form of the calculation in function g. For optimal performance, the program needs to take account of the processor cache size.

```
1  #define INNER 1000
2  #define OUTER 2000
3  #define STRIP_WIDTH 32
4
5  extern int b[OUTER],
6             c[INNER];
7
8  void g(void)
9  {
10 for (int i=0; i < OUTER; i++)
11    for (int j = 0; j < INNER; j++)
12       b[i] += c[j];
13 }
14
15 void h(void)
16 {
17 for (int j = 0; j < INNER; j+=STRIP_WIDTH)
18    for (int i=0; i < OUTER; i++)
19       for (int k=j; k < MIN(INNER, j+STRIP_WIDTH-1); k++)
20          b[i] += c[k];
21 }
```

When the calculation within the loop involves both one- and two-dimensional arrays, the order of loop nesting can depend on how the arrays are indexed. The following code gives an example where the two outer loops are in the opposite order suggested by spatial locality, but temporal locality is the dominant factor here.

```
1  #define INNER 1000
2  #define OUTER 2000
3  #define STRIP_WIDTH 32
4
5  extern int b[OUTER],
6             c[INNER][OUTER];
7
8  void h(void)
9  {
10 /*
11  * for (int i=0; i < INNER; i++)
12  *    for (int j = 0; j < OUTER; j++)
13  *       b[j] += c[i][j];
14  *
15  * is transformed in to the following:
16  */
17 for (int j = 0; j < OUTER; j+=STRIP_WIDTH)
18    for (int i=0; i < INNER; i++)
19       for (int k=j; k < MIN(OUTER, j+STRIP_WIDTH-1); k++)
20          b[k] += c[i][k];
21 }
```

An extensive discussion of the issues involved in taking account of the cache, when ordering array accesses within loops is given in Chapter 9 of Allen and Kennedy.[17]

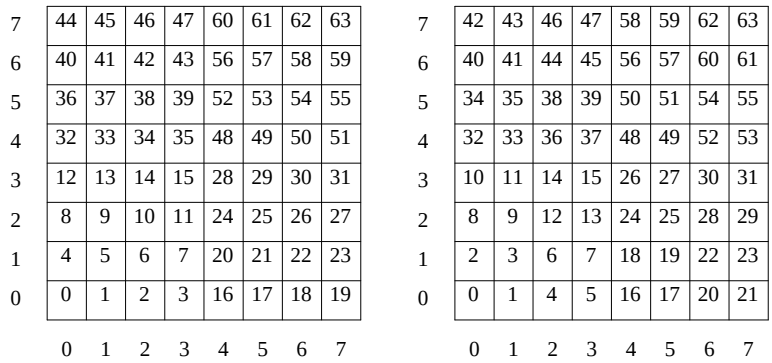


Figure 994.3: Two possible element layouts of an $8 * 8$ array; Blocked row-major layout (left) and Morton element layout (right). Factors such as efficiency of array index calculation, whether array size can be made a power of two, or array shape (e.g., non-square) drive layout selection.[1348]

Provided a multi-dimensional array is accessed through indexing (i.e., not via a pointer whose value has been explicitly calculated) an implementation can use any algorithm it chooses to layout elements in storage. One such algorithm is Morton layout,[1474] which has the advantage that element access performance is independent of whether the elements are accessed in row-major or column-major order, see Table 994.1.

Table 994.1: Cache hit-rate for sequentially accessing, in row-major order, a two-dimensional array stored using various layout methods. If the same array is accessed in column-major order the figures given in the Row-major and Column-major columns are swapped and the Morton layout figure remains unchanged. These figures ignore the impact that accessing other objects might have on cache behavior, and so denote the best hit-rate that can be achieved. Based on Thiyagalingam et al.[1349]

Cache size	Row-major	Morton	Column-major
32 byte cache line	75%	50%	0%
128 cache byte	93.75%	75%	0%
8K byte cache page	99.9%	96.875%	0%

Coding Guidelines

As the preceding discussion in Common implementations showed, the optimal nesting of loops is not always obvious. If all of the elements of a multidimensional array are to be accessed, some choice has to be made about the order in which the associated loops are nested. There is no evidence to suggest that always using the same ordering, rather than say varying the ordering, can lead to a worthwhile reduction in the cost of comprehending source. The use of multidimensional arrays is rarely seen outside of applications that perform some kind of numerical analysis, where developers are often willing to invest effort in tuning performance (and these coding guidelines are not intended to provide efficiency recommendations).

---

995 EXAMPLE Consider the array object defined by the declaration

```
int x[3][5];
```

Here **x** is a 3 × 5 array of **int**s; more precisely, **x** is an array of three element objects, each of which is an array of five **int**s. In the expression **x[i]**, which is equivalent to **(*((x)+(i)))**, **x** is first converted to a pointer to the initial array of five **int**s. Then **i** is adjusted according to the type of **x**, which conceptually entails multiplying **i** by the size of the object to which the pointer points, namely an array of five **int** objects. The results are added and indirection is applied to yield an array of five **int**s. When used in the expression **x[i][j]**, that array is in turn converted to a pointer to the first of the **int**s, so **x[i][j]** yields an **int**.

---

996 **Forward references:** additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.5.2).

6.5.2.2 Function calls

Constraints

function call

The expression that denotes the called function[78)] shall have type pointer to function returning **void** or returning an object type other than an array type. 997

Commentary

function designator converted to type

An occurrence of a function designator in an expression is automatically converted to pointer-to function type.

function declarator return type identifier is primary expression if

A declaration of a function returning an array type is a constraint violation. Since identifiers denoting functions must be declared before they are referenced, the restriction on the object type being other than an array type is superfluous. Declaring a function to return a structure or union type, which contains a member having an array type, is sometimes used to get around the constraint on returning arrays from functions.

C++

5.2.2p3 *This type shall be a complete object type, a reference type or the type* ***void****.*

Source developed using a C++ translator may contain functions returning an array type.

Other Languages

Many languages usually distinguish between a *function* that has a return type and a *procedure* (or *subroutine*), which has no return type (they do not have an explicit **void** type). Some languages restrict functions to returning scalar types, while others allow functions to return any types, including function and array types.

function call arguments agree with parameters

If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. 998

Commentary

That is, the prototype is visible at the point of call. It is not enough for it simply to exist in some translated file.

C++

C++ requires that all function definitions include a prototype.

5.2.2p6 *A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, . . . 8.3.5) than the number of parameters in the function definition (8.4). [Note: this implies that, except where the ellipsis (. . .) is used, a parameter is available for each argument.]*

A called function in C++, whose definition uses the syntax specified in standard C, has the same restrictions placed on it by the C++ Standard as those in C.

Other Languages

Those languages that support some form of function declaration that includes parameter information, often require that the number of arguments in the call agree with the declaration. Some languages (e.g., C++ and Ada) support default values for parameters. In these cases it is possible to omit arguments from the function call.

Coding Guidelines

function declaration use prototype 1810.1

If the guideline recommendation specifying use of function prototypes is followed, all called functions will have a prototype in scope.

argument type may be assigned

Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter. 999

Commentary

The assignment of the argument value to the parameter is similar to an initializer for an object definition. The qualifier is ignored. It only has meaning inside the body of the function that defines the parameter list. Developers sometimes use the term *assignment compatible*, however, the standard does not define this term.

1649 initializer initial value

C++

The C++ language permits multiple definitions of functions having the same name. The process of selecting which function to call requires that a list of viable functions be created. Being a viable function requires:

> *. . . , there shall exist for each argument an implicit conversion sequence that converts that argument to the corresponding parameter . . .* 13.3.2p3

A C source file containing a call that did not meet this criteria would cause a C++ implementation to issue a diagnostic (probably complaining about there not being a visible function declaration that matched the type of the call).

Other Languages

Other languages that contain some kind of type qualifier usually have a rule similar to C.

Semantics

1000 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call.

operator ()

Commentary

Many developers do not think of `()` as an operator. In theory, a single token would be sufficient to indicate a function call (and the same for the `[]` operator). However, existing practice in other languages, and the bracketing effect of using two tokens (it removes the need to use parentheses in those cases where there is more than one parameter), were more important considerations.

An identifier having pointer-to function type that is not followed by parentheses returns the value of the pointer. No function is called.

The first step in improving the execution time of a program is often to measure how much time is spent in each function. Such measurements may or may not include the time spent in any functions that are themselves called by each function (some tools try to provide an estimate of the amount of time spent in any nested calls). Such measurements provide no context information; for instance, the time spent in function `D` may differ between the call chains `A`⇒`B`⇒`D` and `X`⇒`Y`⇒`D`. Call path refinement[535] provides measurements for each set of call paths. The amount of information gathered is much greater and more detailed, but it does allow special cases to be detected (which could then be appropriately specialized), such as the different call paths to `D`.

Understanding the relationship between functions in terms of which one calls, or is called, by another one provides a useful means of comprehending the structure of a program. A call graph is a practical way of displaying this calling/called information. The term *call graph* is usually taken to mean the static call graph. Information on the calls is obtained without executing the program. (Tools that build dynamic call graphs also exist.)

call graph

The C language contains two features that complicate the building of a call graph— a preprocessor and the ability to assign functions and later call them via the object assigned to.

The preprocessor is sometimes used to hide implementation details, such as calls to system libraries, that developers are not expected to be aware of, treating the macro invocation as if it were a function call. At other times a macro is used for efficiency reasons, and any function calls in its body are likely to be of interest to the developer. The preprocessor can also be used to provide conditional compilation. A function call may appear in the program image if certain macros are defined during translation. This raises the question of whether a call graph should only include function calls that are visible in the unpreprocessed source code, or

should it only include function calls that appear in the token stream after preprocessing, or some combination of the two?

When a function is called through an object, having a pointer-to function type, the number of functions that could possibly be called is likely to be greater than one. Depending on the sophistication of the call graph analysis tool, it might show all functions having the pointed-to type of the object, all functions assigned to the object, or those functions that could possibly be pointed to at a given call site (although a recent study[933] was able to obtain some precise results on some of the GNU tools, using relatively inexpensive analysis).

An empirical study of static call graph tools by Murphy, Notkin, Griswold, and Lan[981,982] found that the call graphs they built were all different (in the set of called functions they each created). All tools individually listed called-functions that were not listed as called by any of the other tools, no call graph being a proper subset of another. One tool looked at the source code prior to preprocessing, one tool extracted information from program images that had been built with debugging information switched on, and the other tools operating in various in other ways.

C90

The C90 Standard included the requirement:

> *If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration*
>
> *extern int identifier();*
>
> *appeared.*

A C99 implementation will not perform implicit function declarations.

Other Languages

The use of `()` as the symbol, or operator, indicating a function call is widely used in computer languages. In some languages function calls are implicitly indicated by the type of an identifier and the context in which it occurs; use of `()` are not necessary. Other languages (e.g., Fortran) require the keyword **`call`** to appear before subroutine calls, but not before function calls that appear in an expression. In Lisp the name of the function being called appears as the first expression inside the parentheses. In functional languages it is even possible to invoke a function with an incomplete list of arguments, resulting in the partial evaluation of the called function. A few languages (e.g., ML) allow the function name to appear as an infix operator.

There is a standard for procedure calling: ISO/IEC 13886:1996 *Information technology— Language Independent Procedure Calling (LIPC)*. Quoting from the Introduction: "The purpose of this International Standard is to provide a common model for language standards for the concept of procedure calling.".

Common Implementations

Implementation limits 804

The Implementation limits clause does not specify any minimum requirements on the depth of nested function calls. On most implementations the maximum function-call depth is set by the amount of storage available for the stack to grow into. Some processors have dedicated call stacks and the maximum nesting depth can be as low as 20.[1128]

As processors have continued to evolve, the techniques for calling functions have come full circle. They started out simple, got very complex, and RISC brought them back to being simple (now they are starting to get complicated again). The complex processor instructions of the late 1970s and 1980s tried to perform all of the housekeeping actions associated with a function call. In practice, compiler writers often found it difficult to map the language semantics onto these complex instructions.[1000.1] In many cases it was possible to use alternative instructions that executed more quickly. (Some profiling studies showed that the complex instructions were rarely, if ever, encountered during program execution[1232,1460]). The RISC approach

[1000.1] Measurements of assembly language usage by developers[292] has found that they use a subset of the instructions available to them.

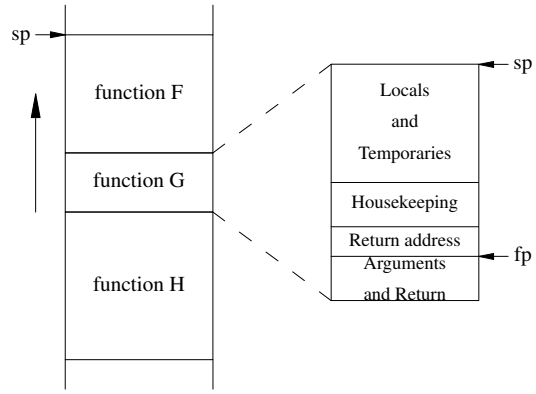


Figure 1000.1: Common storage organization of a function call stack.

simplified everything, going back to very basic support for function calls (which is what low cost processors had been doing all the time, the resources needed to implement complex instructions being too costly).

The minimum requirement for a function-call machine code instruction is that it alters the flow of control (so that the processor starts executing the instructions of the called function), and that there is a mechanism for returning to the instructions after the call (so that execution can resume where it left off).

Processors use a variety of techniques for saving the return address. Some push it onto a stack or a specified storage location (the caller takes responsibility for saving the value); others load it into a register (the called function takes responsibility for saving it). The requirement to call functions recursively restricts the possible mechanisms that can be used to save the return address. An analysis by Miller and Rozas[940] showed that allocating storage on a stack to hold information associated with function calls was more time efficient than allocating it on the heap (even when the cost of garbage collection was minimal).

1026 function call recursive

In most implementations the housekeeping needed to store the arguments is performed in the function performing the call, and the housekeeping needed to allocate storage for locals is handled by the called function. A very common implementation technique is to use a function-call stack, each function call taking storage from this stack to satisfy its requirements, when called, and freeing it up on return. Because function calls properly nest, all operations are performed on the current top of stack.

The address of the start of the stack storage for a function is known as its *frame pointer*. It is often held in a register and a *register + offset* addressing mode is used to access parameters and local objects. The *register + offset* addressing mode is not usually supported by DSP processors,[623] whose only indirect addressing modes are sometimes only pre-/postincrement/decrement of an address register performed as part of a storage access. A number of techniques have been proposed to optimize the generated machine code for processors having these following addressing mode characteristics:

register + offset

- Emulating the *register + offset* addressing mode by using a frame pointer and arithmetic operations to create the address of the object is one possibility[857] (using a peephole optimizer on the resulting code).
- Using a floating frame pointer, arranging the ordering of objects in storage to maximize the likelihood that adjacent accesses in the generated code will be adjacent-locations in storage (so that an increment/decrement will create an address ready for the next access).[375]
- Using algebraic transformations on expression trees to obtain the least cost of access sequence.[1144]
- Genetic algorithms have been proposed[842] as a general solution to the problem. (They are capable of adapting to the different numbers of addressing registers and the different autoincrement ranges available on different processors.)

Another technique is for translators to make use of the characteristics of the applications written for these processors; for instance, recursion is almost unknown, and allocating fixed-storage locations for local objects is often feasible. By building a call graph at link-time, linkers can deduce dependencies between calls and overlay storage for objects, defined in functions, whose functions are not simultaneously active.

The functional programming style of implementing algorithms creates a particular kind of function-call optimization[99] known as *tail calls*. Here the last statement executed in a function is a call to another function (a call to the same function is known as *tail recursion*). Such a recursive call can be replaced by a branch instruction to the start of the function that contains it. This tail recursion optimization idea can be extended to handle calls to other functions, as long as they occur as the last statement executed in a function. A call to another function can be replaced by a branch instruction (it may also be necessary to adjust the stack allocation for the amount of local storage). This optimization saves the creation of a new stack frame by reusing the existing one and can be worthwhile in target environments that have limited storage capacity.

Maintaining a general-purpose stack is an overhead that is not needed for some applications. Vendors of processors designed for use in freestanding environments have come up with a number of alternatives. For instance, the Microchip PIC 18CXX2 processor[931] call instruction pushes the return address onto a dedicated, 31-entry stack. Once this stack is full, the setting of the STVREN (stack overflow reset enable) flag determines the behavior; the processor will either reset or the current top of stack is overwritten with the latest return address. The STKFUL bit will have been set after entry 31 is used, allowing software to detect the pending problem.

One of the methods used by software viruses to gain control of a program is to overwrite a function's stack frame; for instance, changing the return address to point at malicious code that has been copied into some area of storage. A number of techniques to prevent such attacks have been proposed.[294, 443, 1463]

There is often a significant difference in performance overhead between function calls that involve system calls (i.e., calls into the host operating system, such as `fread`, and `fwrite`) and other calls (a factor of 20 has been reported[1137]). Merging multiple system calls into a single call can produce performance improvements.[1137]

Coding Guidelines

cognitive 0 switch

An occurrence of the `()` operator is likely to cause the reader to make at least one cognitive task switch. In some cases (e.g., a call to one of the trigonometric functions) readers may be able to immediately associate a return value with the call and its argument. In more complex cases, readers may have to put some thought into how the arguments passed will affect the return result. They may also need to take into account the fact that the call causes the values of some objects to be modified. There is nothing that can be done at the point of call to reduce the cognitive effort needed for these tasks.

The conditional `if (f)` is sometimes written when `if (f())` had been intended. However, this usage is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines 0 not faults

sequential nesting ()

macro 1933 function-like

The nesting of function calls (i.e., the result of one function call being used as the argument to another function call) raises a comprehension issue. In (which uses function-like macros to reduce the volume of code):

```
1  struct TREE_NODE {
2                    int data;
3                    struct TREE_NODE *left,
4                                     *right;
5                   };
6
7  #define LEFT_NODE(branch) ((branch)->left)
8  #define RIGHT_NODE(branch) ((branch)->right)
9
10 extern struct TREE_NODE *root;
11
12 void f(void)
```

```
{
struct TREE_NODE *leaf;

leaf = LEFT_NODE(RIGHT_NODE(LEFT_NODE(root))).data;
}
```

the expression has to effectively be read from the inside out (given that arguments occur to the right of the function name, the direction of reading is right-to-left). Studies have found that people have difficulty interpreting sentences containing more than two levels of nesting, for instance, "The mouse the cat the dog chased bit died." Speakers of natural language have the same limitations as their listeners, so don't produce such sentences. However, when writing, people do not have to deal with the constraints of realtime communications and consequently tend to write more complex sentences. Writers of source code rarely consider the issue of comprehension by future readers.

1707 statement syntax

Although there are often many ways of phrasing the same natural language sentence ("the mouse died which was bitten by the cat which the dog chased"), the C language rarely offers such flexibility at the expression level. One option is to break an expression into two parts; for instance:

```
left_right = RIGHT_NODE(LEFT_NODE(root));
leaf = LEFT_NODE(left_right).data;
```

An important issue to consider when breaking up an expression into separate components is the application semantics associated with each component. The identifier `left_right` suggests an implementation detail, not a semantic association (the issue of identifier name semantics is discussed elsewhere).

792 Identifier semantic usability

While nested function calls may have a high-comprehension cost, there is no evidence to suggest that in general a guideline recommendation limiting the depth of nesting (and therefore requiring additional code to be written elsewhere) will have a worthwhile benefit (by any reduction in the required cognitive effort in this one case being greater than any increase in effort that occurs elsewhere). Similar human comprehension cost issues affect other operators.

1031 member selection
1095 sequential nesting *

Example

```
typedef int (*p_f)(int);

extern p_f f(float);

void g(void)
{
int loc = f(1.0)(2);
}
```

Usage

How frequent are function calls? The machine code instructions used to call a function may be generated by translators for reasons other than a function call in the source code. Some operators may be implemented via a call to an internal system library routine; for instance, floating-point operations on processors that do not support such operators in hardware. Such usage will vary between processors (see Figure 0.5).

Table 1000.1: Static count of number of calls: to functions defined within the same source file as the call, not defined in the file containing the call, and made via pointers-to functions. Parenthesized numbers are the corresponding dynamic count. Adapted from Chang, Mahlke, Chen, and Hwu.[216]

Name	Within File	Not in File	Via Pointer
cccp	191 (1,414)	4 (3)	1 (140)
compress	27 (4,283)	0 (0)	0 (0)
eqn	81 (6,959)	144 (33,534)	0 (0)
espresso	167 (55,696)	982 (925,710)	11 (60,965)
lex	110 (63,240)	234 (4,675)	0 (0)
tbl	91 (9,616)	364 (37,809)	0 (0)
xlist	331 (10,308,201)	834 (8,453,735)	4 (479,473)
yacc	118 (34,146)	81 (3,323)	0 (0)

Table 1000.2: Percentage of function invocations during execution of various programs in SPECint92. The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* calls to nonleaf functions (the issues surrounding this distinction are discussed elsewhere). The column headed *Direct* lists percentages of calls where a function name appeared in the expression, *Indirect* is where the function address was obtained via expression evaluation. Adapted from Calder, Grunwald, and Zorn.[192]

Program	Leaf	Non-Leaf	Indirect	Direct	Program	Leaf	NonLeaf	Indirect	Direct
burg	72.3	27.7	0.1	99.9	eqntott	85.3	14.7	68.7	31.3
ditroff	14.7	85.3	1.0	99.0	espresso	75.0	25.0	4.0	96.0
tex	20.0	80.0	0.0	100.0	gcc	28.9	71.1	5.4	94.6
xfig	35.5	64.5	6.2	93.8	li	13.4	86.6	2.9	97.1
xtex	50.6	49.4	3.0	97.0	sc	29.1	70.9	0.1	99.9
compress	0.1	99.9	0.0	100.0	Mean	38.6	61.4	8.3	91.7

Table 1000.3: Count of instructions executed and function calls made during execution of various SPECint92 programs on an Alpha AXP21064 processor. *Function calls invoked* includes indirect function calls; *Instructions/Call* is the number of instructions executed per call; *Total I-calls* is the number of indirect function calls made; and *Instructions/I-call* is the number of instructions executed per indirect call. Adapted from Calder, Grunwald, and Zorn.[192]

Program Name	Instructions Executed	Function Calls Invoked	Instructions/Call	Total I-calls	Instructions/I-call
burg	390,772,349	6,342,378	61.6	8,753	44,644.4
ditroff	38,893,571	663,454	58.6	6,920	5,620.5
tex	147,811,789	853,193	173.2	0	–
xfig	33,203,506	536,004	61.9	33,312	996.7
xtex	23,797,633	207,047	114.9	6,227	3,821.7
compress	92,629,716	251,423	368.4	0	–
eqntott	1,810,540,472	4,680,514	386.8	3,215,048	563.1
espresso	513,008,232	2,094,635	244.9	84,751	6,053.1
gcc	143,737,904	1,490,292	96.4	80,809	1,778.7
li	1,354,926,022	31,857,867	42.5	919,965	1,472.8
sc	917,754,841	12,903,351	71.1	13,785	66,576.3
dhrystone	608,057,060	18,000,726	33.8	0	–
Program mean	497,006,912	5,625,468	152.8	397,233	14,614.1

Table 1000.4: Mean and standard deviation of call stack depth during execution of various programs in SPECint92. Adapted from Calder, Grunwald, and Zorn.[192]

Program	Mean	Std. Dev.	Program	Mean	Std. Dev.
burg	10.5	30.84	eqntott	6.5	1.39
ditroff	7.1	2.45	espresso	11.5	4.67
tex	7.9	2.71	gcc	9.9	2.44
xfig	11.6	4.47	li	42.0	14.50
xtex	14.2	4.27	sc	6.8	1.41
compress	4.0	0.07	Mean	12.0	6.29

1001 The postfix expression denotes the called function.

Commentary

The symmetry seen in the `[]` operator does not also apply in the case of the `()` operator. The left operand always denotes the called function.

989 array subscript identical to

An analysis by Zhang and Ryder[1509] found that, for programs using only a single level of pointer-to function indirection (the simplest kind of pointers), statically determining the possible called functions at a call site is NP-hard.

Other Languages

Many languages do not support pointers to functions. For these languages the postfix expression is required to be an identifier.

Coding Guidelines

When the postfix expression is not an identifier denoting the name of a function but an object having a pointer-to function type it can be difficult to deduce which function is being is called. The possibility that the value has been cast from a different pointer-to function type complicates matters even more. Some coding guideline documents adopt a simple solution to the problem of knowing which function is actually being called. They ban the use of nonconstant function pointers.

If the use of nonconstant function pointers is prohibited, what are the alternative constructs available to developers? A sequence of *selection-statement*s could be used to select the function that is to be called. These alternatives do simplify the task of deducing the set of functions that could be called. However, they also potentially increase the effort needed to maintain the source. If any of the functions called changes, it could be necessary to edit more than one location in the source code. (All the **if** statements or **switch** statements referencing the changed function will need to be looked at, unless it is possible to locate the sequence of **if** statements, or **switch** statements in a macro, or inline function.)

1526 inline function

Declaring an array of pointers, initialized with function pointers, creates a single maintenance point in the source.

While use of nonconstant pointers to functions may increase the cognitive effort needed to comprehend source code, the use of alternative constructs could not be said to reduce the effort, and could increase it.

Example

```
1  extern int f_1(void);
2  extern int f_2(void);
3
4  extern int (* const p_f[])(void) = { f_1, f_2 };
5
6  void g(void)
7  {
8  p_f[0]();
9  }
```

Table 1001.1: Static count of functions defined, library functions called, direct and indirect calls to them and number of functions that had their addresses taken in SPECint95. Adapted from Cheng.[222]

Benchmark	Lines Code	Functions Defined	Library Functions	Direct Calls	Indirect Calls	& Function
008.espresso	14,838	361	24	2,674	15	12
023.eqntott	12,053	62	21	358	11	5
072.sc	8,639	179	53	1,459	2	20
085.cc1	90,857	1,452	44	8,332	67	588
124.m88ksim	19,092	252	36	1,496	3	57
126.gcc	205,583	2,019	45	19,731	132	229
130.li	7,597	357	27	1,267	4	190
132.ijpeg	29,290	477	18	1,016	641	188
134.perl	26,874	276	72	4,367	3	3
147.vortex	67,205	923	33	8,521	15	44

The list of expressions specifies the arguments to the function. 1002

Commentary

This defines what the arguments to a C function are.

Usage

function call 289 number of arguments

Usage information on the number of arguments in calls to functions is given elsewhere.

An argument may be an expression of any object type. 1003

Commentary

If the visible function declaration does not include a prototype, this sentence makes passing an argument having an incomplete type undefined behavior. The case where a function prototype is in scope is covered by an earlier constraint.

argument 999 type may be assigned

Only object types have values and a value is needed to assign to the parameter.

Other Languages

Some languages support the use of labels as arguments. While others, particularly functional languages, support the use of types as arguments.

Common Implementations

The undefined behavior of most translators, on encountering an argument that does not have an object type in a call to a function whose declaration does not include a prototype, is to issue a diagnostic. The base document did not support the passing of structure or union types as parameters; a pointer to them had to be passed.

base document 1

Usage

Information on parameter types is given elsewhere (see Table 1831.1).

Table 1003.1: Occurrence of various argument types in calls to functions (as a percentage of argument types in all calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`struct *`	26.8	`void *`	4.0
`int`	16.5	`union *`	3.4
`const char *`	9.7	`unsigned char`	2.5
`char *`	8.4	`enum`	2.1
other-types	8.0	`unsigned short`	1.9
`unsigned int`	7.1	`const void *`	1.8
`unsigned long`	6.3	`long`	1.4

1004 In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[79)]

function call preparing for

Commentary

This form of argument-passing is known as *call by value*. The standard does not specify any order for the evaluation of the arguments.

1025 function call sequence point

In some cases there may not be a parameter to assign the argument to. C supports the passing of variable numbers of arguments in a function call. The C90 Standard introduced the variable arguments mechanism for accessing the values of these arguments. Prior to the publication of C90, a variety of implementation-specific techniques were used by developers. These techniques relied on knowing the argument-passing conventions (which invariably involved taking the address of a parameter and incrementing, or decrementing, walking through the stack to access any additional arguments).

C++

The C++ Standard treats parameters as declarations that are initialized with the argument values:

> *When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument.* 5.2.2p4

The behavior is different for arguments having a class type:

> *A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17).* 12.8p1

This difference has no effect if a program is written using only the constructs available in C (structure and union types are defined by C++ to be class types).

Other Languages

Many languages support a form of argument-passing known as *call by reference*. Here the address of the argument is passed and become the address through which parameter accesses occur. Inside the called function the parameters have the same type as the arguments; they are not pointers to that type. An assignment to the parameter stores the value into the corresponding object passed as the argument. The parameter is effectively an alias for the argument.

A similar, but slightly different, argument-passing mechanism is in/out passing. Here the argument value is copied into the parameter, but modifications to the parameter do not immediately affect the value of the object used as the argument. Just before the function (or procedure) returns, the current value of the parameter is copied into the corresponding object argument. In this case the parameters are not aliases of the corresponding argument objects, but values can still be passed back to the calling function.

Java passes arguments that have primitive type (the C arithmetic type) by value. Arguments having any other type are passed by reference.

Some languages do not (e.g., Haskell, Miranda, sometimes Lisp, Scheme) evaluate the arguments to a function before it is called. They are only evaluated on an as-needed basis, inside the called function, when they are accessed. Such a mechanism is known as *lazy*, or *normal-order evaluation*.

Algol 60 used what is known as *call by name* argument-passing. When this language was designed, developers were familiar with the use of assembly language macros. The arguments to these macros were expanded out in the body of the macro, just like the C preprocessor. The designers of Algol 60 decided that arguments to functions should have the same semantics. Algol 68, and other languages derived from Algol 60, did not duplicate this design decision.

Some languages (e.g., Ada, C++, and Fortran 90) support *named* parameters. Here the name of the parameter appears in the argument list to the left of the argument value. By using names it is possible to list the arguments in any order rather than relying on a default order implicit in the function definition.

Ichbiah, Barnes, Firth, and Woodger[619] discuss the rationale behind the selection of argument-passing mechanisms in Ada.

Common Implementations

One of the simplest, and commonly seen, techniques is to push arguments onto the stack used to hold the function return address and local object definitions. The addresses of each parameter in the called function are the stack locations holding the corresponding argument.

Different methods of passing arguments are seen on different processors and in different operating systems (Johnson[675] discusses the original thinking behind the C calling sequence). In a hosted environment there are usually strong commercial incentives for all translator vendors to use the same conventions (it enables calls to libraries written using different translators and languages to be intermixed). Since the early 1990s it has become common practice for processor vendors to publish a document called an ABI (Applications Binary Interface).[613, 1345–1347, 1517] One of the details specified by this document are the argument-passing conventions.

Calling conventions can also depend on how functions are defined. Functions defined using prototypes and the static storage class are good candidates for aggressive optimization of their calling sequence. Because they are not externally visible, the translator does not have to worry about calls where the prototype might not be visible. In the case of functions that are externally visible, translators have to play safe and follow documented calling conventions (unless using a link-time optimizer[985]).

The IAR PICmicro compiler[612] supports the **__bankN_func** specifier (where N is a decimal literal denoting particular storage banks), which can be used by developers to indicate, to the translator, which storage bank should be used to pass arguments in calls to functions (defined using this specifier).

function call 289
number of arguments

Analysis of function calls shows that it is rare for many arguments to be passed. This usage suggests the optimization of passing the first few arguments in registers and pushing any subsequent arguments onto the stack. In practice this is an optimization, which at first sight looks very simple, but turns out to be potentially very complex and sometimes not even an optimization at all. For instance, if the called function itself calls a function, the contents of the reserved registers need to be saved before the new arguments for the call are loaded into them.

register
function call
housekeeping

There is one task that needs to be performed, before performing a function call, that the standard does not mention. The contents of registers holding values that will be used after the call returns need to be saved. Implementations tend to adopt one of two strategies for saving these registers— caller saves registers (it only needs to save the ones that contain values that are accessed after the call returns) or callee saves registers (it only needs to save the registers it knows it modifies). Caller saves is commonly seen. An analysis by Davidson and Whalley[327] compared various methods and concluded that a hybrid scheme was often the best. It has been proposed that processors include an instruction that, when executed, tells the processor that the contents of a particular register will not be read again before another value is loaded, so called *dead value* information.[899] An overall performance increase of 5% is claimed from not having to save/restore values across function calls or process context switches.

In the 1990s researchers started to investigate link-time optimizations. The linker has information on register usage available to it, by function, across all translation units; thus, it is possible to work out exactly, not make worst-case assumptions, which registers need to be saved and restored. Measurements from one such tool[985] showed context-sensitive interprocedural register liveness analysis reducing the total number of load instructions performed during program execution by 2.5% to 5% (the context-insensitive figure was 2.5–3%).

Processor vendors have also attempted to design instructions that reduce the overhead associated with saving register contents. Some CISC processors[284] include call instructions which automatically save the contents of some registers. An alternative processor architectural technique, adopted by Sun in their SPARC architecture[1450] and more recently by Intel in the Itanium processor family,[1139] is to use overlapping register windows. Here the processor supports a large internal register file (often 128 or more registers) of which a subset (usually 32, the *register window*) are available to the program. Execution of a *call* instruction

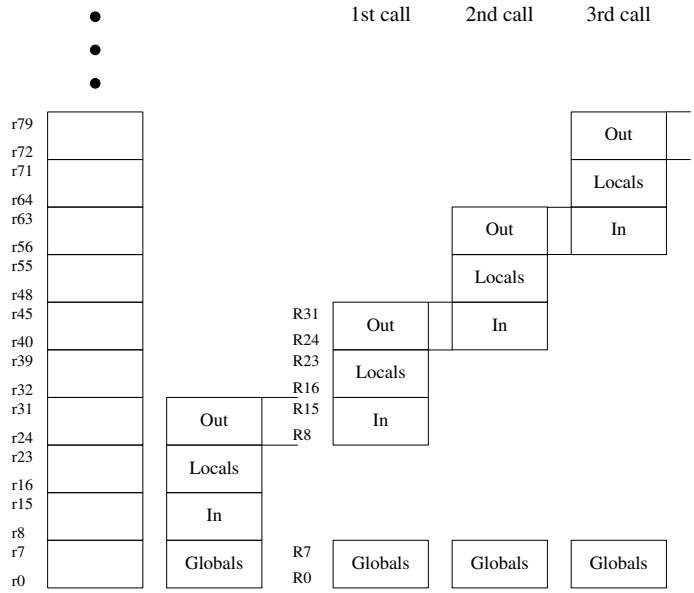


Figure 1004.1: A processor's register file (on the left) and a mapping to register windows for registers accessible to a program, after 0, 1, 2, and 3 *call* instructions have been executed. The mapping of the first eight registers is not affected by the *call* instruction.

causes the register window to slide up the internal register file such that, for instance, 16 *new* registers become available and 16 registers contents are *saved* (instructions always refer to registers numbered between 0 and 31, in the case of registers numbered between 8 and 31 the actual internal register referenced depends on the current depth of call instructions). Up to eight arguments in the *output* registers become available in the called function via its parameters in the *input* registers (see Figure 1004.1). Executing a return instruction slides the register window back, making the previous register contents available again. If there are sufficient nested call instructions the register file fills up and some of its contents have to be spilled to storage (a time consuming process). The performance advantage of registers windows comes from typical program behavior, where call depth does not vary a great deal during program execution (i.e., register file contents rarely have to be spilled and reloaded from storage). The optimal size of register window to use involves trade-offs among several hardware design factors.[461]

1005 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4.

Commentary

This defines the type of the function call expression (to which the **( )** operator is applied) and the type of its value (created by the evaluation of an expression appearing in a **return** statement in the called function's body).

C90

A rather embarrassing omission from the original C90 Standard was the specification of the type of a function call. This oversight was rectified by the response to DR #017q37.

C++

Because C++ allows multiple function definitions having the same name, the wording of the return type is based on the type of function chosen to be called, not on the type of the expression that calls it.

The type of the function call expression is the return type of the statically chosen function . . .

Other Languages

Some languages use the convention that a function call always returns a value, while procedure (or subroutine) calls never return a value.

Common Implementations

Although not explicitly specified in the C90 Standard, all implementations effectively used the above definition (and did not regard the omission of a definition in the standard as implying undefined behavior).

Table 1005.1: Occurrence of various return types in calls to functions (as a percentage of the return types of all function calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`void`	35.8	`union *`	3.2
`int`	30.5	`unsigned long`	3.1
`struct *`	9.1	`char *`	3.1
`void *`	6.3	`unsigned int`	2.1
other-types	5.2	`char`	1.6

Otherwise, the function call has type **`void`**. 1006

Commentary

In this case it has no value.

C++

C++ also supports functions returning reference types. This functionality is not available in C.

---

function result attempt to modify

If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined. 1007

Commentary

The result referred to here is any actual storage returned by the function call. This situation can only occur if the function return type is a structure or union type (see Examples that follow).

C90

This sentence did not appear in the C90 Standard and had to be added to the C99 Standard because of a change in the definition of the term lvalue. lvalue 721

C++

The C++ definition of lvalue is the same as that given in the C90 Standard, however, it also includes the wording:

5.2.2p10 *A function call is an lvalue if and only if the result type is a reference.*

In C++ it is possible to modify an object through a reference type returned as the result of a function call. Reference types are not available in C.

Common Implementations

The implementation of functions that return a structure or union type usually involves the use of, translator allocated, temporary storage. This storage is usually allocated in the stack frame of the calling function and passed to the called function as an additional, hidden from the developer, argument. The C Standard only requires that the lifetime of the temporary storage used to hold the return value exist until the next sequence point after the return. Reuse of any part of this temporary storage by developers (e.g., to hold other temporary values) could overwrite any previous contents.

Coding Guidelines

By continuing to access the result of a function call, developers are relying on undefined behavior for what is essentially an efficiency issue (they are manipulating the returned result directly rather than assigning it to a declared object). However, on the basis that occurrences of this usage are likely to be rare, a guideline recommendation is not considered cost effective.

Example

In the following the temporary used to hold the value returned by `f` may, or may not be used for other purposes after the function has returned.

```
1  struct S {
2            int m1;
3            unsigned char m2[10];
4           };
5
6  extern struct S f(void);
7
8  void g(void)
9  {
10 int *pi = &(f().m1); /* Get hold of the address of part of the result. */
11 unsigned char uc_1,
12               uc_2;
13
14 *pi = 11; /* Undefined behavior. */
15
16 /*
17  * In the following there is a sequence point after the comma operator.
18  */
19 uc_1 = f().m2[uc_2=0, 2]; /* Also undefined, and probably surprising to readers. */
20
21 /*
22  * Depending on the unspecified order of evaluation, the
23  * following may also be undefined.
24  */
25 uc_1 = f().m2[f().m1];
26 }
```

1008 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **float** are promoted to **double**.

called function no prototype

Commentary

This sentence applies when an old-style function declaration is visible at the point of call. If no prototype is visible, there is no information on the expected type of the argument available to the translator. On many processors the minimum-size object that can be pushed onto the stack (the most common argument-passing mechanism) has type **int**. The integer promotion rules for arguments match how their value is likely to be passed to the function, even if they had a type whose size was smaller than **int**. This behavior also matches the case where an argument expression contains operators, causing the integer promotions to be performed on the operands.

485 int natural size

A consequence of performing the integer promotions on arguments is that the translator needs to generate matching machine code for accessing the parameters in the called function. It is possible that it may need to implicitly convert a value back to its original type.

1011 function definition ends with ellipsis

The conversion of real type **float** to **double** does not apply to complex types. The following wording was added to the Rationale by the response to DR #206:

Rationale

float _Complex is a new type with C99. It has no need to preserve promotions needed by pre-ANSI-C. It does not break existing code.

C++

In C++ all functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

Other Languages

Most languages do not contain multiple integer types. Those languages that do contain multiple floating-point types do not usually specify that arguments having these types need be converted prior to the call.

Coding Guidelines

function declaration use prototype 1810.1

If the guideline recommendation specifying use of function prototype declarations is followed the integer promotions will not be performed.

---

default argument promotions

These are called the *default argument promotions*. 1009

Commentary

integer promotions 675

This defines the term *default argument promotions*. They are a superset of the integer promotions in that they also promote the type **float** to **double**.

C++

The C++ Standard always requires a function prototype to be in scope at the point of call. However, it also needs to define default argument promotions (Clause 5.2.2p7) for use with arguments corresponding to the ellipsis notation.

Coding Guidelines

The term *argument promotions*, or the *arguments are promoted*, are commonly used by developers. There is no obvious benefit in investing effort in changing this existing usage.

---

arguments same number as parameters

If the number of arguments does not equal the number of parameters, the behavior is undefined. 1010

Commentary

Prior to the C90 Standard no prototypes using the ellipsis notation were available. However, developers still expected to be able to pass variable numbers of arguments to functions. The C Committee added the ellipsis notation for passing variable numbers of arguments to functions. However, the Committee could not make it a constraint violation for the number of arguments passed to disagree with the number of parameters in a nonprototype function definition. It would have invalidated a large body of existing source code.

Other Languages

Most languages require that the number of arguments agree with the number of parameters. Some classes of languages (e.g., functional) are specifically intended to support function invocations where fewer arguments than parameters are passed.

Common Implementations

Passing the incorrect number of parameters is both a common mistake, a programming technique that continues to be used by some developers, and something that occurs within long-existent source code. Being able to handle this case is usually seen as commercially essential for an implementation.

If the argument-passing conventions are such that the leftmost argument is nearest the top of the stack (if the stack grows down it will have the lowest address, if the stack grows up it will have the highest address), then the address of a particular parameter in the called function does not have a dependency on the number of arguments given in the call. The address of the parameter is known at translation time and can be accessed by indexing off the stack pointer (or the frame pointer, if the implementation uses one, see Figure 1000.1).

There are a number of implementation techniques for passing arguments to functions. These techniques all rely, to some degree, on the caller performing some of the housekeeping. As well as assigning arguments to parameters, this housekeeping also usually includes popping them off the stack when the called function returns (assuming this is the argument-passing convention used).

Coding Guidelines

Given the guideline recommendation dealing with the use of function prototypes this situation is only likely to occur when developers are modifying existing code that does not use function prototypes. Mismatches between the number and types of arguments in function calls, when no prototype is visible, and the corresponding function definition are a very common source of programming error (experience shows that once developers who use the non-prototype form are introduced to the benefits of using prototypes, they rarely want to switch back to using the non-prototype form). These mismatches are not required to be diagnosed by a translator if no prototype is visible at the point of call.

1810.1 function declaration use prototype

46 undefined behavior

Developers do not usually provide the incorrect number of arguments in a function call on purpose (an existing developer practice is to call the Unix system function `open` with 2 or 3 arguments depending on the value of the second). A guideline recommending that the expected number of arguments always be passed serves no purpose. The solution to the underlying problem is to use prototypes. The extent to which it is cost effective to modify existing code to use prototypes is discussed elsewhere.

1810 prototypes cost/benefit

1011 If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined.

function definition ends with ellipsis

Commentary

This C sentence applies when calling a function, defined using a prototype, when an old-style declaration (not the prototype) is visible at the point of call.

When translating a function whose definition uses a prototype an implementation knows the types of the parameters and can make use of this information. In particular it need not implicitly convert a reference to a parameter that does not have a type that is the same as its promoted type. It may also choose to assign different storage addresses to such parameters, when declared with and without the use of prototype notation.

1019 function call prototype visible

There are advantages to requiring implementations to make sure this case is well defined. (It would enable developers to slowly migrate their source code over to using prototypes, changing definitions to use prototypes, and eventually getting around to ensuring that a prototype is visible from all calls to that function.) The disadvantage of such an approach is that it would have tied implementations to backward compatibility, removing the possibility of most of the optimizations that prototypes enable translators to perform.

The occurrence of an ellipsis in a function's definition does not automatically require an implementation to guarantee that all calls to that function have well-defined behavior. In some implementations the argument/parameter mechanism used when a function is defined using ellipsis is completely different from when an ellipsis is not used. If, at the point of call, it is not known that the function definition contains an ellipsis (because no prototype is visible), the generated code may be incorrect.

Other Languages

Most languages either specify that some form of prototype declarations always be used, or do not provide any mechanism for specifying the types of parameters. However, it is not uncommon for a language to evolve into supporting both kinds of function definition (provided it originally supported the nonprototype form only). For instance, the ability to declare subroutines taking arguments was added in release 5.003 of Perl.

Common Implementations

There are two commonly seen cases: (1) the parameter has an integer type whose rank is less than that of **`int`**; (2) the type of the argument, after promotion, is not compatible with the parameter type. It is not uncommon for implementations to give unexpected results when reading a parameter having a character type in a function defined using a prototype, when that function has been called in a context where no prototype

is visible. In some cases the machine code generated by a translator for the called function, accesses the value of the parameter using an instruction that reads more than eight bits (the typical number of bits in a character type). The assumption being made by the translator is that at the point of call the argument will have been converted to the character type, bringing it into the representable range of that type (the typical implementation behavior for all character types). If no prototype is visible at the point of call, this conversion will not have occurred, and it is possible for a value outside of the representable range of a character type to be assigned to the parameter.

Assuming that arguments are passed by pushing values onto a stack, the number of bytes pushed onto the stack will depend on the number of bytes in the promoted argument type. If the types **int** and **long**, for instance, have the same size, then mixing their usage in arguments is likely to have no surprising results. However, if these two types have different sizes, it is likely that after an argument having the other type is evaluated, any subsequent arguments (which will depend on the order of evaluation) will be pushed at addresses that are different from where their corresponding parameters expect them to be.

Coding Guidelines

function declaration use prototype 1810.1

If a function is defined using a prototype, the only reason for not having this prototype visible at the point of call is that the translator being used does not support prototypes (i.e., it is different from the one that translated the function definition). The guideline recommendation dealing with the use of function prototypes is applicable here.

Example

The handling of parameters inside a function definition can be different from when a prototype is used than when it is not used. In:

file_1.c

```
int f_np(a, b)
signed char a;
int b;
/*
 * Function body assumes that any call will perform the default argument
 * promotions.  So it will insert casts, where necessary, on parameters.
 */
{
return a + b;
}

int f_pr(signed char a, int b)
/*
 * Function body assumes that a prototype is visible at the point of call.
 * Hence the arguments will have been cast at the point of call.  So there
 * is no need to insert an implicit conversion on any parameter reference.
```

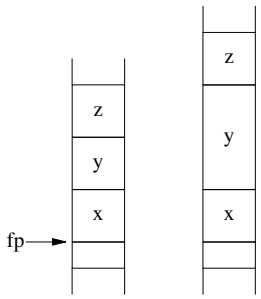


Figure 1011.1: An example of the impact, on relative stack addresses, of passing an argument having a type that occupies more storage than the declared parameter type. For instance, the offset of z, relative to the frame pointer *fp*, might be changed by passing an argument having a type different from the declared type of the parameter y (this can occur when there is no visible prototype at the point of call to cause the type of the argument to be converted).

```
 */
{
return a + b;
}
```

file_2.c

```
extern int f_np();
extern int f_pr();

extern signed char sc;
extern int i;
extern int glob;

int g(void)
{
glob = f_np(sc, i);
glob = f_pr(sc, i);
}
```

at the call to `f_np` there is no visible information on the function parameter types. If there had been, the first argument would have been cast to type **`signed char`**.

So what is the value of the parameter `a` inside the function `f_pr`? No implicit conversion need occur. If machine code to load one byte is generated by the translator, the value will only ever be in the range of the type **`signed char`**. However, if the generated code loads a larger unit of storage, it may be quicker to load a word, making the assumption that the top bits are all zero rather than to load a byte and zeroing the top bits. In the load word case the possible range of values is that of type **`int`**, not **`signed char`**.

1012 If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

argument in call incompatible with function definition

Commentary

The rationale for this compatibility requirement is the same as that for the prototype case (the only difference is that the compatibility check is against the promoted type of the parameter). The call and definition need to agree on how the type of the argument affects how and where values are passed to the called function.

1011 function definition ends with ellipsis

C90

The C90 Standard did not include any exceptions.

C++

All functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

Common Implementations

The undefined behavior for these cases of all known C90 implementations is as defined in C99.

Coding Guidelines

Passing an argument having the incorrect type is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. The two cases listed in the following C sentences are intended to codify practices that are relatively common in existing code, not for use in newly written code. Adhering to the guideline recommendation on using function prototypes enables translators to implicitly perform any necessary conversion or to issue a diagnostic is none is available.

0 guidelines not faults

1810.1 function declaration use prototype

Example

Here both the function definition and the point of call must agree on how they treat the type of the argument. In the following:

file_1.c
```
#include <stdio.h>

void f(a)
short a;
{
printf("a=%d\n", a);
}
```

file_2.c
```
extern void f();

void g(void)
{
f(10);  /* First call. */
f(10L); /* Second call. */
}
```

the argument passed in the first call to `f` is compatible with the promoted type of the parameter in the definition of `f`. In the second call the argument is not compatible, even though the value is representable in the promoted parameter type (and also the unpromoted parameter type).

---

footnote 78

78) Most often, this is the result of converting an identifier that is a function designator. 1013

Commentary

A self-evident, to experienced developers, piece of information is carried over from the C99 document.

C++

The C++ language provides a mechanism for operators to be overloaded by developer-defined functions, creating an additional mechanism through which functions may be called. Although this mechanism is commonly discussed in textbooks, your author suspects that in practice it does not account for many function calls in C++ source code.

Other Languages

This statement is true in most programming languages.

Common Implementations

The implicit conversion of a function designator to a pointer-to function is optimized away by most implementations, generating a machine code instruction to perform a function call to a known (at link-time) address.

Usage

In most programs an identifier is converted in more than 99% of cases, although a lower percentage is occasionally seen (see Table 1001.1).

---

footnote 79

79) A function may change the values of its parameters, but these changes cannot affect the values of the arguments. 1014

Commentary

operator () 1000

The storage used for parameters is specific to the function invocation. The final value of a parameter is never copied back to the corresponding argument. The storage used for parameters is released when the invocation of the function, that defines, them returns.

C++

The C++ reference type provides a call by address mechanism. A change to a parameter declared to have such a type will immediately modify the value of its corresponding argument.

This C behavior also holds in C++ for all of the types specified by the C Standard.

Other Languages

In languages that support various forms of argument passing (e.g., call by value, call by reference, or **OUT** style parameters) changes to the value of a parameter may or may not affect the value of the argument, or may be a violation of the language semantics (e.g., parameters defined using **IN** in Ada).

1004 function call preparing for

Coding Guidelines

Some coding guideline documents do not permit parameters to be modified within the function definition. They are considered to be read-only objects. The rationale for such a guideline often draws parallels with other languages, where a modification of a parameter also modifies the corresponding argument. Should a set of C coding guidelines take into account the behavior seen in other languages, and if so, which others languages need to be considered? Would a guideline recommendation against modifying parameters reduce or increase faults in programs? would it reduce or increase the effort needed to comprehend a function? Are the alternatives more costly than the original problem, if there is one? Without even being able to start answering these questions, there is no point in attempting to create a guideline recommendation.

1015 On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to.

Commentary

Parameters can have any object type (which include pointer types). Assigning the value of an argument, having a pointer type, to a parameter, having a pointer type, has the same effective semantics as assigning any pointer value to an object. Both pointers point at the same object. A modification of the value of the pointed-to object through either pointer reference will be visible via the other pointer reference.

C++

This possibility is not explicitly discussed in the C++ Standard, which supports an easier to use mechanism for modifying arguments, reference types.

Other Languages

This statement is generally true in languages that support pointers. In some languages (e.g., function languages) it is never possible to have two pointers (or references as they are usually known) to the same object. An assignment, or argument, would cause the pointed-to object to be copied. The pointer assigned would then point at a different object that had the same value.

Coding Guidelines

One of the issues overlooked by coding guideline documents which recommend against the use of pointers, is that argument-passing in C uses pass by value. If pointers cannot be used, the only way for a function to pass information back to its caller is via a returned value, or by modifying file scope objects. The alternatives could be worse than what they are replacing.

Usage

Pointer types are the most commonly occurring kind of parameter type (see Table 1831.1).

1016 A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Commentary

This adjustment is made to the types of the parameters in a function definition. The arguments are similarly adjusted.

1821 function definition syntax
729 array converted to pointer
732 function designator converted to type

Coding Guidelines

The benefit of defining parameters using array types rather than the equivalent pointer type comes from the additional information provided (the expected number of elements) to tools that analyze source code (enabling them to potentially be more effective in detecting likely coding faults). There is new functionality in C99 that allows developers to specify a lower bound on the number of elements in the object passed as an argument and to specify that after conversion, the pointer rather than the pointed-to object is qualified.

1599 function declarator static

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types; 1017

Commentary

Many argument expressions are integer constants, often having type **signed int**. A requirement that all such arguments be explicitly cast, or contain a suffix, in those cases where the corresponding parameter has type **unsigned int** was considered onerous. This exception allows unsuffixed integer constants to be used as arguments where the corresponding parameter has type **unsigned int**.

This exception requires that the rank of the signed and unsigned integer types be the same. It does not provide an exception for the case where an integer constant argument has a different type from the corresponding parameter type, but is representable in the parameter type. There are two requirements specified elsewhere that ensure implementations meet this requirement: (1) identical values of corresponding signed/unsigned integer types, and (2) storage and alignments requirement.

signed 486
integer
corresponding
unsigned integer
footnote 509
31

Other Languages

Most languages do not contain both signed and unsigned integer types.

Coding Guidelines

Most integer constants that appear in source code have small positive values (98% of decimal literals and 88% of hexadecimal literals are less than 32,768) of type **signed int** (only 2% of constants are suffixed). In these cases there is no obvious benefit to using a cast/suffix when the corresponding parameter has an unsigned type. For other kinds of arguments, more information on the cost/benefit of explicit casts/suffixes for arguments is needed before it is possible to estimate whether any guideline recommendation is worthwhile.

integer 825
constant
usage

Example

In the following values of type **unsigned int** and **int** are passed as arguments. Prior to C99, the C90 Standard specified that this was undefined behavior.

```
1  extern unsigned int ui;
2
3  extern void f();
4
5  void g(void)
6  {
7  f(ui);
8  f(22);
9  }
```

— both types are pointers to qualified or unqualified versions of a character type or **void**. 1018

Commentary

Requiring that arguments having type pointer-to unqualified character type be explicitly cast to the parameter type (to remove an undefined behavior that another part of the standard specifies, although not normatively) was regarded as being excessive. A commonly used argument of type pointer-to unqualified character type is a string literal. Some functions in the C library have parameters whose type is `const char *`. A large amount of existing code uses the pre-C Standard definition of these functions, which does not include the **const** qualifier.

footnote 565
39
Normative 18
references

Other Languages

Some languages have generic pointer types that can be passed as arguments where the corresponding parameter type is a pointer to some other type.

Coding Guidelines

What kind of pointer argument types and pointer parameter types, in an old-style function definition, are likely to occur in existing code? It is unlikely that parameters having a pointer type will be qualified (these were introduced into C at the same as prototypes), and they are probably more likely to be pointers to character type than pointer to **void**. The only constant pointer type is the null pointer constant, whose type is not under developer control. If this exception case is invoked, the argument type is likely to either have a pointer-to qualified type or be a pointer to **void**.

More information on the cost/benefit of explicit casts, for arguments, is needed before it is possible to evaluate whether any guideline recommendation is worthwhile.

1019 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type.

function call prototype visible

Commentary

When a prototype is visible, argument conversion occurs at the point of call. (For functions defined without a prototype, parameters are converted inside the function definition.) A constraint requires that the conversion path exist. The implicit conversion rules, for assignment, simply specify that the value being assigned is converted to the object assigned to. There are constraints that ensure that this conversion is possible.

1011 function definition ends with ellipsis
999 argument type may be assigned
1303 simple assignment

C90

The wording that specifies the use of the unqualified version of the parameter type was added by the response to DR #101.

C++

The C++ Standard specifies argument-passing in terms of initialization. For the types available in C, the effects are the same.

> *When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument.* 5.2.2p4

Other Languages

This is how call by value is usually specified to work in languages that support some form of function prototypes.

Common Implementations

Passing an argument is not always the same as performing an assignment. In the case of arguments, the storage used to pass the value may be larger than the type being passed because of alignment requirements. For instance, arguments may always be passed in storage units that start at an even address. If characters occupy 8 bits, the unused portion of the parameter storage unit may need to be set to zero.

Coding Guidelines

The issues involved in any implicit conversion of arguments are the same as those that apply in other contexts.

653 operand convert automatically

1020 The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter.

Commentary

Type conversion does not stop; as such, there are no available parameter types to convert to. However, the standard does specify another rule that may cause argument type conversions (see following C sentence).

C++

There is no concept of starting and stopping, as such, argument conversion in C++.

Other Languages

Common Lisp supports functions taking a variable number of arguments. However, it uses dynamic typing and it is not necessary to perform type checking and conversion during translation.

Common Implementations

Some implementations treat the arguments corresponding to the ellipsis notation differently from other arguments. The ellipsis notation makes it very difficult to specify, in advance, optimal passing conventions for those arguments. Some implementations put the arguments corresponding to the ellipsis notation in a dedicated block of storage that makes them easy to access using the `va_*` macros. Other implementations simply push these arguments onto the stack, along with all other arguments.

The default argument promotions are performed on trailing arguments. 1021

Commentary

default argument promotions called function no prototype 1009 1008

The rationale for performing the default argument promotions on trailing arguments is the same as that for performing them on arguments of calls to functions where no prototype is visible. There is also the added benefit of simplifying the implementation of the `va_*` macros by permitting them to assume that these promotions have occurred; for instance, an implementation can rely on an object having type **`short`** having been converted to type **`int`** when passed as a trailing argument. These macros are specified to have undefined behavior if one of their arguments has a type that is not compatible with its promoted type.

Coding Guidelines

Some developers incorrectly assume that because a prototype is visible the arguments corresponding to the ellipsis notation are passed as is; that is, no implicit conversions are applied to them at the point of call (this assumption only leads to a fault is a certain combinations of events occur). Other than developer training, there is no obvious worthwhile guideline recommendation.

Example

```
1  extern unsigned char uc;
2
3  extern void f(int, char, float, ...);
4
5  void g(void)
6  {
7  f(1, uc, 1.2F, 99, 3.4F);
8  f(1, uc, 1.2F, 5.6, 'w', 0x61, 'y');
9  }
```

No other conversions are performed implicitly; 1022

Commentary

This C sentence specifies what happens when there is no function prototype in scope at the point of call.

C++

In C++ it is possible for definitions written by a developer to cause implicit conversions. Such conversions can be applied to function arguments. C source code, given as input to a C++ translator, cannot contain any such constructs.

Common Implementations

base document 1

The base document required that arguments having a structure or union type be converted to pointers to those types. This conversion was performed because the passing of arguments having structure or union types was not supported. It was assumed that, if an argument had such a type, a pointer to it had been intended and the translator implicitly took its address.

1023 in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

Commentary

No comparison occurs even if the function definition, that does not use a prototype, is contained in the same source file as the call and occurs lexically before it in that file. The number and types of arguments are only compared with those of the parameters in a function definition if it uses a prototype and occurs lexically prior to the call in the same source file. In all other cases it is the visible declaration that controls what checks are made on the arguments in the call. The C Standard does not require any cross-translation unit checks against function definitions.

C++

All function definitions must include a function prototype in C++.

Other Languages

This behavior is common in languages that do not have a construct similar to a function prototype declarator. Without any parameter information to check against, there is little that implementations can do. The behavior for the case where a function call appears in the source textually before a definition of the function occurs varies between languages. Some (e.g., Algol 68) require that argument and corresponding parameter types be compared, which requires the implementation to operate in two passes, while others (e.g., Fortran) have no such requirement.

Common Implementations

Translators and static analysis tools sometimes perform checks on the arguments to functions that have been defined using the old-style definition. This checking can include the following:

- *Comparing the number and type of arguments against the corresponding parameters.* This can be done if the called function is defined in the same source file as the call, or by performing cross translation unit checks during translation phase 8.
- *Comparing the types of the arguments in different calls to the same function.* Often more than one call to the same function is made from the same source file. Flagging differences in argument types between these different calls provides a consistency check, although it will not detect differences between argument types and their corresponding parameter types.

1024 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

function call not compatible with definition

Commentary

766 pointer to function converted 1611 function compatible types

For this situation to occur either a pointer-to function type has to be explicitly cast to an incompatible pointer to function type, or the declaration visible at the point of call is not compatible with the function definition it refers to. A cast could occur at the point of call, or when a function designator is assigned to a pointer-to function type.

Note: Calling a function, where the visible declaration does not include a prototype, with a particular set of argument types does not affect the type pointed to by the expression that denotes the called function (there is no creation of a composite type).

C++

In C++ is it possible for there to be multiple definitions of functions having the same name and different types. The process of selecting which function to call requires that a list of viable functions be created (it is possible that this selection process will not return any matching function).

It is possible that source developed using a C++ translator may contain pointer-to function conversions that happen to be permitted in C++, but have undefined behavior in C.

Common Implementations

Implementations usually treat such a call like any other. They generate machine code to evaluate the arguments and perform the function call. Unexpected behavior is likely to occur if the argument types differ from the corresponding parameter types and the two types have different sizes.

Differences in the return type can result in unexpected behavior, depending on the implementation technique used to handle function return values. In some implementations the return value, from a function, is passed back in a specific register. A difference in function return type can result in the value being passed back in a different register (or a storage location). Other implementations pass the return value back to the caller on the stack. A difference in the number of bytes actually occupied by the return value and the number of bytes it is expected to occupy can result in a corrupt stack (which often holds function return addresses) leading to completely unpredictable program behavior.

stack 449

Coding Guidelines

Intentionally calling a function defined with a type that is not compatible with the pointed-to function type, visible at the point of call, is at best relying on an implementation's undefined behavior being predictable (for argument-passing and values being returned). This usage may deliver different results if the source is translated using different implementations, or even different versions of the same implementation. If this usage occurs within existing code, it potentially ties any retranslation of that source to the translator originally used.

There is a programming technique that relies on the arguments in a function call depending on the context in which they occur. For instance, an array of pointers to function may have elements pointing at more than one function type.

```
1  extern int f0(), f2(), f4();
2  extern int f1(), f3();
3
4  int (*(a[5]))() = {f0, f1, f2, f3, f4};
5
6  void g(int p)
7  {
8  if (p % 2)
9     a[p](1, 2);
10 else
11    a[p](1);
12 }
```

ellipsis 1601
supplies no information
identifier 422.1
declared in one file
identifier 422.2
with extern linkage
reference shall #include

An alternative technique that does not rely on implementation-defined behavior is to declare the various `f` functions and the array element type using a function prototype that contains an ellipsis. Adhering to the guideline recommendations on defining an external declaration in one source file, which is visible at the point where its corresponding function is defined prevents a visible function declaration from being incompatible with its definition.

---

function call sequence point

The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call. 1025

Commentary

The sequence point before the call guarantees that any side effects in the evaluation of the arguments, or the function designator, will have occurred prior to the call. When a value is returned, there is also a sequence point immediately before the called function returns.

return expression 1719
sequence point

C++

5.2.2p8

All side effects of argument expression evaluations take effect before the function is entered.

While the requirements in the C++ Standard might appear to be a subset of the requirements in the C Standard, they are effectively equivalent. The C Standard does not require that the evaluation of any other operands, which may occur in the expression containing the function call, have occurred prior to the call and the C++ Standard does not prohibit them from occurring.

Other Languages

A few languages do specify an evaluation order for the arguments in a function call; for instance, Java specifies a left-to-right order.

Common Implementations

Most implementations evaluate the arguments in the order that they are pushed onto the call stack, with the function designator evaluated last (this behavior usually minimizes the amount of temporary storage needed for argument evaluation). The most common order for pushing arguments onto the call stack implies a right-to-left evaluation order. Using this order makes it possible to pass variable numbers of arguments, the address of the last argument (the rightmost one in the argument list) in a call to a function may not be known when its definition is translated. However, the address of the first argument, in the argument list, can always be made the same relative to the frame pointer of the called function by pushing it last (see Figure 1000.1). By being pushed last the value of the first argument is always closest to the local storage, and successive arguments further way.

Coding Guidelines

This unspecified behavior is a special case of an issue covered by a guideline recommendation. Function calls are sometimes mapped to function-like macro invocations. In this case an argument that contains side effects may be evaluated more than once (in the body of the macro replacement). This issue is discussed elsewhere.

187.1 sequence points
all orderings give same value
1939 macro
arguments separated by comma

Example

```
1  #include <stdio.h>
2
3  int glob = 0;
4
5  int f(int value, int *ptr_to_value)
6  {
7  if (value != *ptr_to_value)
8     printf("first argument evaluated before assigned to glob\n");
9  if (value == *ptr_to_value)
10    {
11    printf("value == pointer to value\n");
12    /*
13     * Between the previous equality test and here glob may
14     * have been assigned the value 1 (in main).  Let's check...
15     */
16    if (value != *ptr_to_value)
17       printf("value != pointer to value\n");
18    }
19 return 1;
20 }
21
22 int main(void)
23 {
24 (glob = 1) + f(glob, &glob);
25 }
```

function call recursive

Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions. 1026

Commentary

object 272 storage outside function image

This is a requirement on the implementation. It is supported by a restriction on where storage for objects can be allocated. The standard does not specify any minimum limit on the depth of nested function calls that an implementation must support.

Other Languages

Some languages do not require implementations to support recursive function calls. Fortran (prior to Fortran 90) and some dialects of Basic do not support recursive calls (although some implementations of Fortran did allow recursion). Some languages (e.g., Fortran 90, and later PL/1 and CHILL) require that functions called recursively be declared using the keyword **`recursive`**.

Common Implementations

The implication of this requirement is that objects defined locally within each function definition must be allocated storage outside of the sequence of machine instructions implementing a function.

Recursive function calls are rare in most applications. Some implementations make use of this observation to improve the quality of generated machine code. The default behavior of the implementation being to assume that a program contains no recursive function calls. If a function is not called recursively, it is not necessary for the implementation to allocate storage for its local objects every time it is called— storage for these local objects could be allocated in a global area at a known, fixed address; these addresses can be calculated at link-time using the program's call tree to deduce which function lifetimes are mutually exclusive. Objects local to a set of functions whose lifetimes are mutually exclusive, can be overlaid with each other, minimizing the total amount of storage that is required.

Removing the need to support recursion removes the need to maintain a stack of return addresses. One alternative sometimes used is to store a function's return address within its executable code (ensuring that the surrounding instructions jump around this location).

Coding Guidelines

Many coding guideline documents ban recursive function calls on the basis that their use makes it impossible to calculate, statically, a program's maximum storage and execution requirements. (This is actually an overstatement; the analysis is simply very difficult and only recently formalized.[128, 129]) However, in some mathematical applications, recursion often occurs naturally and sometimes requires less effort to comprehend than a nonrecursive algorithm. Also, recursive algorithms are often easier to mathematically prove properties about than their equivalent iterative formulations.

Rather than provide a guideline recommendation and matching deviation, these coding guideline subsections leaves it to individual projects to handle this special case, if necessary.

EXAMPLE In the function call 1027

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions **`f1`**, **`f2`**, **`f3`**, and **`f4`** may be called in any order. All side effects have to be completed before the function pointed to by **`pf[f1()]`** is called.

Commentary

An order of execution could be guaranteed by using the comma operator:

```
1  (t4=f4(), t3=f3(), t2=f2(), t1=f1(),  (*pf[t1]) (t2, t3 + t4))
```

Common Implementations

Because its value is needed last, the function `f1` is likely to be called last.

1028 **Forward references:** function declarators (including prototypes) (6.7.5.3), function definitions (6.9.1), the **return** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1029 The first operand of the . operator shall have a qualified or unqualified structure or union type, and the second operand shall name a member of that type.

operator . first operand shall

Commentary

The . operator effectively operates on the left operand to make its member names visible. The right operand indicates which one is to be accessed.

Other Languages

This notation is common to most languages that support some form of structure or union type. Cobol uses the keyword **OF** and reverses the order of the operands.

Common Implementations

The definition originally used in K&R allowed the right operand to belong to any visible structure or union type. This restricts member names to being unique; there was a single structure and union name space, not a separate one for each type. A member name was effectively treated as a synonym for a particular offset from a base address.

An extension sometimes seen in translators[717] for freestanding environments is to define the behavior when the left operand has an integer type and the right operand is an integer constant. In this case the . operator returns the value of the bit indicated by its right operand from the value of the left operand. Other extensions, seen in both environments, include anonymous unions and even unnamed members.[1350] In this case the right operand may be any member contained within the type of the left operand (translator documentation does not always specify the rules used to disambiguate cases where more than one member could be chosen).

```
1  void Plan_9(void)
2  {
3  typedef struct lock {
4                        int locked;
5                       } Lock;
6  struct {
7          union {
8                 int u_mem1;
9                 float u_mem2;
10               }; /* Constraint violation, but supported in some implementations. */
11         int s_mem2;
12         Lock;    /* Constraint violation, but supported in some implementations. */
13        } obj;
14
15 obj.u_mem1=3; /* Access member without any intermediate identifiers. */
16 obj.locked=4;
17 }
```

1030 The first operand of the -> operator shall have type "pointer to qualified or unqualified structure" or "pointer to qualified or unqualified union", and the second operand shall name a member of the type pointed to.

Commentary

The -> operator is really a shorthand notation that allows (*p).mem to be written as p->mem. Because many structure members' accesses occur via pointers this notational convenience makes for more readable source code.

Other Languages

Many languages do not need to provide a shorthand notation covering this case. In those languages where the indirection operator appears to the right of the operand, member selection does not require parentheses. In Pascal, for instance, `R.F` selects the field `F` from the record `R`, while `P_R^.F` selects the field from the record pointed to by `P_R`. However, it might be asked why it is necessary to use a different operator just because the left operand has a pointer type. The language designers could have chosen to specify that an extra indirection is performed, if the left operand has a pointer type. The Ada language designers took this approach (to access all of a pointed-to object the reserved word **`all`** needs to be used).

Semantics

member selection

A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. 1031

Commentary

union 1427 members start same address

Members designated using the `.` operator are at a translation time known offset from the base of the structure (members of unions always have an offset of zero). This translation-time information is available via the `offsetof` macro.

Other Languages

Cobol and PL/1 support elliptical member references. One or more selection operators (and the corresponding member name) can be omitted, provided it is possible for a translator to uniquely determine a path to the specified member name. By allowing some selection operators to be omitted, a long chain of selections can be shortened (it is claimed that this improves readability).

Cobol and Algol 68 use the reverse identifier order (e.g., `name OF copper` rather than `copper.name`).

Coding Guidelines

Structure or union types defined in system headers are special in that development projects rarely have any control over their contents. The members of structure and union types defined in these system headers can vary between vendors. An example of the different structure members seen in the same structure type is provided by the `dirent` structure. The POSIX.1 Standard[636] requires that this structure type include the members `d_name` and `d_namelen`. The Open Groups Single Unix Specification[1487] goes further and requires that the member `d_ino` must also be present. Looking at the system header on Linux we find that it also includes the members `d_off` and `d_type`; that is

```
1  struct dirent {
2                  ino_t d_ino;                /* SUS */
3                  off_t d_off;                /* Linux */
4                  unsigned char d_type;       /* Linux */
5                  unsigned short int d_reclen; /* POSIX.1 */
6                  char d_name[256];           /* POSIX.1 */
7                 };
```

While developers cannot control the availability of members within structure types on different platforms, they can isolate their usage of those members that are vendor-specific. Encapsulating the use of specific members within functions or macro bodies is one solution. Most host development environments predefine a number of macros and these can be used as feature test macros by a development group.

feature test macro 96

Table 1031.1: Number of member selection operators of the same object (number of dot selections is indicated down the left, and the number of indirect selections across the top). For instance, `x.m1->m2` is counted as one occurrence of the dot selection operator with one instance of the indirect selection operator. Based on the translated form of this book's benchmark programs.

. \ ->	0	1	2	3	4	5
0	0	165,745	10,396	522	36	4
1	28,160	34,065	3,437	230	7	0
2	3,252	6,643	579	26	0	0
3	363	309	35	5	0	0
4	16	33	2	0	0	0
5	0	15	0	0	0	0

1032 The value is that of the named member [DR283)], and is an lvalue if the first expression is an lvalue.

Commentary

A member access that reads a value behaves the same as an access that reads from an ordinary identifier.

The number of constructs whose result has a structure, or union, type that is not an lvalue has been reduced to one in C99. The cast operator does not return an lvalue, but its operand cannot be a structure or union type.

The issue of members having a bit-field type is discussed elsewhere.

721 lvalue
1131 footnote 85
1134 cast scalar or void type
672 bit-field in expression

Common Implementations

The base address of file scope objects having a structure type may not be known until link-time. The offset of the member will then need to be added to this base address. Nearly all linkers support some kind of relocation information, in object files, needed to perform this addition at link-time. The alternative is to generate machine code to load the base address and add an offset to it, often a slower instruction (or even two instructions).

Example

```
1  struct s {
2            int m;
3           };
4
5  void f(void)
6  {
7  int *p_i;
8  struct s l_s;
9
10 p_i = &(l_s.m);
11 /*
12  * A strictly conforming program can only cast an object having
13  * a structure type to exactly the same structure type.
14  */
15 p_i = &(((struct s)l_s).m); /* Constraint violation, & not applied to an lvalue. */
16 }
```

1033 If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

struct qualified
result qualified

Commentary

The qualifiers on the result type are the union of the set of type qualifiers on the two operands.

Other Languages

The logic of an outer qualifier applying to all the members of a structure or union applies to qualifiers followed in most programming languages.

Example

```
1  void f(void)
2  {
3  const struct {
4                 const int mem_1;
5                 int mem_2;
6                } x;
7
8  x.mem_1; /* Has type const int, not const const int */
9  x.mem_2; /* Also has type const int */
10 }
```

EXAMPLE 1040 qualifiers Also see a C Standard example elsewhere.

A postfix expression followed by the -> operator and an identifier designates a member of a structure or union object. 1034

Commentary

A consequence of the pointer dereference operator, *, being a prefix rather than a postfix operator and having a lower precedence than the member selection operator, ., is that it is necessary to write (*p).mem to access a member of a pointed-to structure object. To provide a shorthand notation, the language designers could either have specified that one level of pointer indirection is automatically removed if the left operand of the . operator has pointer type (as Ada and Algol 68 do), or they could specify a new operator. They chose the latter approach, and the -> operator was created.

C++

The C++ Standard specifies how the operator can be mapped to the dot (.) form and describes that operator only.

5.2.5p3 *If E1 has the type "pointer to class X," then the expression E1->E2 is converted to the equivalent form (*(E1)).E2; the remainder of 5.2.5 will address only the first option (dot)[59].*

Common Implementations

In this case the base address of the pointed-to object is not usually known until program execution, although the offset of the member from that address is known during translation. Processors invariably support a *register+offset* addressing mode, where the base address of an object (the address referenced by the left operand) has already been loaded into register. It is common for more than one member of the pointed-to object to be accessed within a sequence of statements and this base address value will compete with other frequently used values to be held in a register. register + offset 1000

The time taken to load a value from storage can be significant and processors use a variety of techniques to reduce this performance overhead. Walking tree-like data structures, using the -> operator, can result in poor locality of reference and make it difficult for a processor to know which storage locations to prefect data from. A study by Luk and Mowry[872] investigated translator-controlled prefetching schemes. cache 0

Coding Guidelines

The issues common to all forms of member access are discussed elsewhere. There is a difference between the use of the . and -> operators that can have coding guideline implications. Accessing the member of a nested structure type within one object can lead to a series of successive . operators being applied to the result of preceding . operators. In this case the sequence of operators is narrowing down a particular member within a larger object; the member selections are intermediate steps toward the access of a particular member. It is likely that different members will have different types and if an incorrect member appears in the selection chain, a diagnostic will be issued by a translator. member selection 1031

The -> operator is often used to access members of a tree-like data structure whose nodes may have a variety of types and are independent objects. A sequence of -> operators represents a path to a particular node in this tree, relative to other nodes. The member selection path used to access a particular member is not limited to the depth of nesting of the structure types involved. To work out what the final node accessed represents, developers need to analyze the path used to reach it, where each node on the path is often a separate object. (An incorrect path is very unlikely to generate a diagnostic during translation)

In many cases developers are not interested in the actual path itself, only what it represents. When there is a sequence of -> operators in the visible source code, developers need to deduce, or recognize, the algorithmic semantics of the operation sequence. This deduction can require significant developer cognitive resources. There are a number of techniques that might be used to reduce the total resources required, or reduce the peak load required. The following example illustrates the possibilities:

```
typedef struct TREE_NODE *TREE_PTR;
struct TREE_NODE {
                  int data;
                  TREE_PTR left,
                           right;
                  };

#define LEFT_NODE(branch) ((branch)->left)
#define RIGHT_NODE(branch) ((branch)->right)
#define Get_Road_Num(node) ((node)->left->right->left.data)

extern TREE_PTR root;

void f(void)
{
int leaf_valu;
TREE_PTR temp_node_1,
         temp_node_2,
         destination_addr,
         road_name;

leaf_valu = root->left->right->left.data;
leaf_valu = ((root->left)->right)->left.data;
/*
 * Break the access up into smaller chunks.  Is the access
 * above easier or harder to understand than the one below?
 */
temp_node_1 = root->left;
temp_node_2 = temp_node_1->right;
leaf_valu = temp_node_2->left.data;
/*
 * If the intermediate identifier names had a name that suggested
 * an association with what they represented, the effort needed,
 * by developers, to create their own associations might be reduced.
 */
destination_addr = root->left;
road_name = destination_addr->right;
leaf_valu = road_name->left.data;
/*
 * Does hiding the underlying access details, an indirection
 * operator, make any difference?
 */
leaf_valu = LEFT_NODE(RIGHT_NODE(LEFT_NODE(root))).data;
/*
 * The developer still has to deduce the meaning of the access from the path
 * used.  Why not hide the complete path behind a meaningful identifier?
 */
leaf_valu = Get_Road_Num(root);
```

```
49  }
```

sequential nesting () 1000
sequential nesting * 1095

Some of these issues also affect other operators. At the time of this writing there is insufficient experimental evidence to enable a meaningful cost/benefit analysis to be performed.

The value is that of the named member of the object to which the first expression points, and is an lvalue.[80)] 1035

Commentary

Unlike uses of the `.` operator, there are no situations where a pointer can refer to an object that is not an lvalue (although many implementations' internal processing of the `offsetof` macro casts and dereferences `NULL`).

lvalue 721

bit-field in expression 672

The issue of members having a bit-field type is discussed elsewhere.

If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member. 1036

Commentary

struct 1033
qualified result qualified
effective type 948

Qualification is based on the pointed-to type. The effective type of the object is not considered. The qualifiers on the result type are the union of the set of type qualifiers on the two operands.

---

union special guarantee

One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. 1037

Commentary

alias analysis 1491

value stored in union 586

At one level this guarantee is codifying existing practice. At another level it specifies a permission, given by the standard to developers, which a translator needs to take account of when performing pointer alias analysis. This guarantee is an exception to the general rule that reading the value of a member of a union type after a value has been stored into a different member results in unspecified behavior. It only applies when two or more structure types occur within the definition of the same union type. Being contained within the definition of a union type acts as a flag to the translator; pointers to objects having these structure types may be aliased.

Two or more structure types may share a common initial sequence. But if these types do not appear together within the same union type, a translator is free to assume that pointers to objects having these two structure types are never aliases of each other.

The existing C practice, which this guarantee codifies, is an alternative solution to the following problem. The structure types representing the nodes of a tree data structure (or graph) used by algorithms, sometimes have a number of members in common and a few members that differ. To simplify the processing of these data structures a single type, representing all members, is usually defined. The following example shows one method for defining this type:

```
1   struct REC {
2              int kind_of_node_this_is;
3              long common_2;
4              int common_3;
5              union {
6                     float differ_1;
7                     short differ_2[5];
8                     char *differ_3;
9                    } unique;
10             };
```

Use of a type built in this fashion can be inflexible. It creates a dependency between developers working with different types. The type will generally be contained in a single file with any modifications needing to

go through a change control mechanism. There is also the potential issue of wasted storage. One of the union members may use significantly more storage than any of the other members. Allocating storage based on the value of `sizeof(struct REC)` could be very inefficient. An alternative solution to defining a common type is the following:

```
struct REC_1 {
            int kind_of_node_this_is;
            long common_2;
            int common_3;
                  float differ_1;    /* Members that differ start here. */
            };
struct REC_2 {
            int kind_of_node_this_is;
            long common_2;
            int common_3;
                  short differ_2[5]; /* Members that differ start here. */
            };
struct REC_3 {
            int kind_of_node_this_is;
            long common_2;
            int common_3;             /* Members that differ start here. */
                  char *differ_3;
            };

union REC_OVERLAY {
                  struct REC_1 rec_1;
                  struct REC_2 rec_2;
                  struct REC_3 rec_3;
                  };
```

In this case the different structures (`REC_1`, `REC_2`, etc.) could be defined in different source files, perhaps under the control of different developers. When storage is allocated for a particular kind of node, the type appearing as the operand of **`sizeof`** can be the specific type that is needed; there is no wasted storage caused by the requirements of other nodes.

The wording " . . . anywhere that a declaration of the complete type of the union is visible." is needed to handle the following situation:

afile.c

```
#include <stdio.h>

union utag {
            struct tag1 {
                        int m1;
                        double d2;
                        } st1;
            struct tag2 {
                        int m1;
                        char c2;
                        } st2;
            } un1;

extern int WG14_N685(struct tag1 *pst1, struct tag2 *pst2);

void f(void)
{
if (WG14_N685(&un1.st1, &un1.st2))
   printf("optimized\n");
else
   printf("unoptimized\n");
}
```

tag 634 declared with same

If the two structure declarations did not appear within the same union declaration in the following translation unit, a translator would be at liberty to perform the optimization described earlier.

bfile.c

```
union utag {
            struct tag1 {
                        int m1;
                        double d2;
                        } st1;
            struct tag2 {
                        int m1;
                        char c2;
                        } st2;
            } un1;

int WG14_N685(struct tag1 *pst1, struct tag2 *pst2)
{
pst1->m1 = 2;
pst2->m1 = 0;   /* Could be an alias for pst1->m1 */
return pst1->m1;
}
```

C90

The wording:

> *anywhere that a declaration of the complete type of the union is visible.*

was added in C99 to handle a situation that was raised too late in the process to be published in a Technical Report. Another wording change relating to accessing members of union objects is discussed elsewhere.

EXAMPLE 1044 member selection value 586 stored in union

C++

Like C90, the C++ Standard does not include the words " . . . anywhere that a declaration of the complete type of the union is visible."

Other Languages

This kind of guarantee is unique to C (and C++). However, other languages support functionality that addresses the same problem. Two languages commonly associated with the software development of applications requiring high-integrity, Pascal and Ada, both support a form of the union type within a structure type supported by C.

Common Implementations

All implementations known to your author assign the same offset to members of different structure types, provided the types of all the members before them in the definition are compatible.

Coding Guidelines

The idea of overlaying members from different types seems to run counter to traditional thinking on how to design portable, maintainable programs. The C Standard requires an implementation to honor this guarantee so its usage is portable. The most common use of common initial sequences is in dynamic data structures. These involve pointers and pointers to different structure types which are sometimes cast to each others types. There can be several reasons for this usage, including:

- Sloppy programming, minimal change maintenance, or a temporary fix that becomes permanent; for instance, some form of tree data structure whose leafs have some structure type. An enhancement requires a new list of members representing completely different information, and rather than add these members to the existing structure, it is decided to create a new structure type (perhaps to save space). Changing the type of the member that points at the leafs, to have a union type (its members

having the types of the two structures), would require changing accesses to that member to include the name of the appropriate union member. Existing code could remain unchanged if the existing member types remained the same. Any references to the new structure type is obtained by casting the original pointer type.

- Support for subtyping, inheritance, or class hierarchies— constructs not explicitly provided in the C language.

The following discussion is based on the research of Stiff et al.,[1301, 1302] and others.[215] In:

```
typedef struct {
                 int x,
                     y;
                } Point;

void update_position(Point *);

typedef enum { RED, BLUE, GREEN } Color;

typedef struct {
                 int x,
                     y;
                 Color c;
                } Color_Point;
typedef struct {
                 Point where;
                 Color c;
                } CW_Point;
typedef struct {
                 char base[sizeof(Point)];
                 Color c;
                } CA_Point;

void change_color(Color_Point *);

void f(void)
{
Point a_point;
Point *some_point;
Color_Point a_color_point;
CW_Point a_c_point;
/* ... */
update_position((Point *)&a_color_point);
update_position(&a_c_point.where);
/* ... */
some_point = (Point *)&a_color_point;    /* An upcast.  */
/* ... */
change_color((Color_Point *)some_point); /* A downcast. */
}
```

the types `Color_Point`, `CW_Point` and `CA_Point` can be thought of as subtypes of `Point`, one having the same common initial sequence as `Point` (guaranteeing that the members `x` and `y` will have the same offsets) and the other making use of its definition. Both forms of definition are found in source code.

1038 common initial sequence

The issue is not about how the contents of structures are defined, but about how their usage may rely on layout of their members in storage. For instance, developers who depend on member layout to reinterpret an object's contents. In the above example a call to `update_position` requires an argument of type `Point *`. The design of the data structures for this program makes use of the fact that all members called `x` and `y` have the same relative offsets within their respective objects.

530 structure type sequentially allocated objects

Commonly seen object-oriented practice is to allow objects of a derived class to be treated as if they were objects of a base class. Sometimes called an *upcast* (because a subtype, or class, is converted to a type or class). A conversion in the other direction, known as a *downcast*, is seen less often.

Discussing how members in one structure might occupy the same relative (to the start of the structure) storage locations as members in another structure would normally bring howls of protest about unsafe practices. Rephrasing this discussion in terms of the object-oriented concepts of class hierarchies and inheritance lends respectability to the idea. The difference between this usage in C and C++ (or Java) is that there is no implicit language support for it in the former case; the details have to be looked after by the developer. It is this dependency on the developer getting the layout correct that is the weakest link.

Duplicating sequences of members within different structures, as in the declaration of `Color_Point`, violates the single location principle, but has the advantage of removing the need for an additional dot selection in the member reference. Using a single declared type within different structures, as in `CW_Point`, is part of the fabric of normal software design and development, but has the disadvantage of requiring an additional member name to appear in an access.

Stiff et al. base the definition of the term *physical type* on the layout of the members in storage. The form of the declarations of the members within the structure is not part of its physical type. Upcasting does leave open the possibility of accessing storage that does not exist; for instance, if a pointer to an object of type `Point` is converted to a pointer to an object of type `Color_Point`, and the member `c` is subsequently accessed through this pointer.

Usage

Measurements by Stiff et al.[1301] of 1.36 MLOC (the SPEC95 benchmarks, gcc, binutils, production code from a Lucent Technologies' product and a few other programs) showed a total of 23,947 casts involving 2,020 unique types. For the `void *` ⇔ `struct *` conversion, they found 2,753 upcasts (610 unique types), 2,788 downcasts (606 unique types), and 538 cases (60 unique types) where there was no matching between the associated up/down casts. For the `struct *` ⇔ `struct *` conversions, they found 688 upcasts (78 unique types), 514 downcasts (66 unique types), and 515 cases (67 unique types) where there was no relationship associated with the types.

---

common initial sequence

Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members. 1038

Commentary

This defines the term *common initial sequence*.

The rationale given for the same representation and alignment requirements of signed/unsigned types given in footnote 31 does not apply to a common initial sequence. Neither is a special case called out, like that for arguments. Also the rationale given in footnote 39 for pointer to **`void`** does apply to a common initial sequence. There are additional type compatibility rules called out for pointer to **`void`** and other pointer types for some operators, but there are no rules called out for common initial sequences.

The requirement that successive members of the same structure have increasing addresses prevents an implementation from reordering members, in storage, to be different from how they appear in the source (unless it can deduce that such a reordering will not affect the behavior of the program). The common initial sequence requirement thus reduces to requiring the same amount of padding between members.

footnote 509 31
argument 1012 in call incompatible with function definition
footnote 565 39
member 1422 address increasing
structure 1424 unnamed padding

C++

3.9p11 *If two types T1 and T2 are the same type, then T1 and T2 are layout-compatible types.*

Two POD-structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

POD is an acronym for *Plain Old Data* type.

C++ requires that types be the same, while C requires type compatibility. If one member has an enumerated type and its corresponding member has the compatible integer type, C can treat these members as being part of a common initial sequence. C++ will not treat these members as being part of a common initial sequence.

Common Implementations

In choosing the offset from the start of the structure of each structure member, all implementations known to your author only consider the type of that member. Members that appear later in the structure type do not affect the relative offset of members that appear before them. Translators that operate in a single pass have to assume that the initial members of all structure types might be part of a common initial sequence. In the following example the body of the function `f` is encountered before the translator finds out that objects it contains have access types that are part of a common initial sequence.

10 implementation single pass

```
1  struct t_1 {
2              int mem_1;
3              double mem_2;
4              char mem_3;
5             } g_1;
6  struct t_2 {
7              int mem_4;
8              double mem_5;
9              long mem_6;
10            } g_2;
11
12 void f (void)
13 { /* Code accessing g_1 and g_2 */ }
14
15 union {
16        struct t_1 mem_7;
17        struct t_2 mem_8;
18       } u;
19
20 /* ... */
```

Coding Guidelines

A common initial sequence presents a number of maintenance problems, which are all derived from the same requirement— the need to keep consistency between textually different parts of source code (and no translator diagnostics if there is no consistency). The sequence of preprocessing tokens forming a common initial sequence could be duplicated in each structure definition, or the common initial sequence could be held in its own source file and **`#include`**d in all the structure type definitions that define the members it contains. Since the cost/benefit of the different approaches is unknown, and there is no established existing practice that addresses the software engineering issues associated with reliably maintaining a consistent common initial sequence, no recommendations are made here.

Example

```
1  typedef short int BUCKET;
2
3  struct R_1 {
4              short int mem_1;
5              int mem_2 : 1+1;
6              unsigned char mem_3;
7              };
```

```
8   struct R_2 {
9               short mem_1;
10              int mem_2 : 2;
11              short mem_3; /* Not a common initial sequence member. */
12              };
13  struct R_3 {
14              short int member_1;
15              int mem_2 : 3-1;
16              long mem_3;
17              };
18  struct R_4 {
19              BUCKET mem_1;
20              int mem_2 : 1+1;
21              float mem_3;
22              };
23  union node_rec {
24                  struct R_1 n_1;
25                  struct R_2 n_2;
26                  struct R_3 n_3;
27                  struct R_4 n_4;
28                  };
```

EXAMPLE 1 If **f** is a function returning a structure or union, and **x** is a member of that structure or union, **f().x** is a valid postfix expression but is not an lvalue. 1039

Commentary

lvalue 725
converted to value

It is not an lvalue because it occurs in a context where an lvalue is converted to a value.

EXAMPLE
qualifiers

EXAMPLE 2 In: 1040

```
struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;
```

the various members have the types:

```
s.i         int
s.ci        const int
cs.i        const int
cs.ci       const int
vs.i        volatile int
vs.ci       volatile const int
```

Commentary

derived type 557
qualification
qualified 1492
array type

Other examples are given elsewhere.

footnote
80

80) If **&E** is a valid pointer expression (where **&** is the "address-of" operator, which generates a pointer to its operand), the expression **(&E)->MOS** is the same as **E.MOS**. 1041

Commentary

Similarly, if `P` is a valid pointer expression, the expression `(*P).MOS` is the same as `P->MOS`. The term *address-of* is commonly used by developers to denote the **&** operator.

C++

The C++ Standard does not make this observation.

footnote DR283

1042 DR283) If the member used to access the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning").

Commentary

The issue accessing overlapping union members and *type punning* is discussed elsewhere.

An object representation is the sequence of bytes that is interpreted, using a type, to form a value.

The member being accessed may include bytes that are not within the object last written to. These bytes may have an indeterminate or an unspecified value.

This footnote was added by the response to DR #283.

531 union type overlapping members
531 type punning union
574 object representation
461 object initial value indeterminate
586 value stored in union

1043 This might be a trap representation.

Commentary

The same object representation may be a value representation when interpreter using one type but a trap representation when interpreted using another type (e.g., a value of type **long** may be a trap representation when its object representation is reinterpreted as the type **float**). Accessing a trap representation causes undefined behavior.

This footnote was added by the response to DR #283.

574 object representation
595 value representation
579 trap representation reading is undefined behavior

1044 EXAMPLE 3 The following is a valid fragment:

EXAMPLE member selection

```
union {
        struct {
                int    alltypes;
        } n;
        struct {
                int    type;
                int    intnode;
        } ni;
        struct {
                int    type;
                double doublenode;
        } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
        if (sin(u.nf.doublenode) == 0.0)
                /* ... */
```

The following is not a valid fragment (because the union type is not visible within function **f**):

```
struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
        if (p1->m < 0)
                p2->m = -p2->m;
        return p1->m;
}
int g()
{
        union {
                struct t1 s1;
                struct t2 s2;
        } u;
```

```
        /* ... */
        return f(&u.s1, &u.s2);
}
```

Commentary

The sense of "not a valid fragment" in the second example is unspecified behavior (because different members of a **union** type are being accessed without an intervening store). Given the arguments passed to the function `f`, the likely behavior is to return the absolute value of the member `m`.

value 586 stored in union

C90

In C90 the second fragment was considered to contain implementation-defined behavior.

C++

The behavior of this example is as well defined in C++ as it is in C90.

Common Implementations

An optimizer might assume that the objects pointed at by the parameters of `f` are disjoint (because there is no connection between the structure types `t1` and `t2` at the time the function `f` is first encountered during translation). There is a long history of C translators being able to operate in a single pass over the source code and in some cases the function `f` might be completely translated before moving on to the function `g` (the example could have put `g` in a separate file). Given the availability of sufficient registers, an optimizer (operating in a single pass) may hold the contents of `p1->m` in a different register to the contents of `p2->m`. So, although the member `m` will hold a positive value after the call to `f`, the value returned by both functions could be negative.

implementation 10 single pass

Forward references: address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1). 1045

6.5.2.4 Postfix increment and decrement operators

Constraints

---

postfix operator constraint

The operand of the postfix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue. 1046

Commentary

C99 added complex types to the set of scalar types. However, the Committee did not define a meaning for the postfix increment and decrement operators for the complex types. (A natural result, had polar coordinates been chosen for the representation, would have been to increment or decrement the magnitude; one possibility in a cartesian coordinate representation would be to increment the real part, which is the effect in `C+=1`.)

The postfix `++` operator might be used in conjunction with an object of type **_Bool** to perform an atomic test-and-set operation. However, C99 does not require an implementation to support this usage, even if the Library typedef name `sig_atomic_t`, defined> has type **_Bool**.

C90

The C90 Standard required the operand to have a scalar type.

C++

D.1p1 *The use of an operand of type* ***bool*** *with the postfix* ++ *operator is deprecated.*

Other Languages

These operators are usually seen in languages that claim to be *C-like*.

Common Implementations

Many processors include instructions that are special cases of other instructions. For instance, addition and subtraction instructions that encode a small constant value as part of the instruction (removing the need to load the constant into a register before using it as an operand). The number of bits used for encoding this constant in an instruction tends to be greater for wider instructions. In some cases (usually 8-bit processors) the instruction may be simply an increment/decrement by one.

Looping constructs often involve incrementing or decrementing an objects value by one. The object may be a loop counter or a pointer into storage. Some processors combine a counting operation, comparison of modified value, and branch based on result of the comparison in a single instruction, while other processors support addressing modes that modify the contents of a register (or storage location) as part of the execution of the instruction.[968] In some cases these addressing modes can be used to efficiently access successive elements of an array. To be able to use these specialized instructions the appropriate constructs need to occur in the source and a translator needs to be capable of detecting it. For instance, the Motorola 68000[968] includes a postincrement and a predecrement addressing mode, which are only useful in some cases.

1763 iteration statement syntax

```
while (*p++ == *q++) /* Postincrement */
   ;
while (*--p == *--q) /* Predecrement */
   ;
/* Motorola 68000 addressing modes no use for the following */
while (*++p == *++q) /* Preincrement */
   ;
while (*p-- == *q--) /* Postdecrement */
   ;
```

Coding Guidelines

Incrementing or decrementing the value of an object by one is a relatively common operation. This usage created two rationales for the original inclusion of these postfix operators in the C language. It simplified the writing of a translator capable of generating reasonable quality machine code and it provided a shortened notation (in both the visual and conceptual sense) for developers. Another rationale for the use of these operators has come into being since they were first created— existing practice. Because they occur frequently in existing source developers have acquired extensive experience in recognizing their use in source. Their presence in existing code ensures that future developers will also gain extensive experience in recognizing their use. These operators are part of the culture of C.

912 punctuator syntax

The issue of translator optimization increment/decrement operators is not as straight-forward as it might appear; it is discussed elsewhere. The conclusions reached are even more applicable to the postfix operators because of their additional complexity.

1082 prefix ++ incremented

From the coding guideline perspective use of these operators can be grouped into the following three categories:

1. *The only operator in an expression statement.* In this context the result returned by the operation is ignored. The statement simply increments/decrements its operand. Use of the postfix, rather than the prefix, form follows the pattern seen at the start of most visible source code statement lines— an identifier followed by an operator (see Figure 940.2). For this reason the postfix operators are preferred over prefix operators, an expression statement context.

1082 prefix ++ incremented

2. *One of the operators in a full expression that contains other operators.* Experience shows that developers' strategy of scanning down the left edge of the source looking for objects that are modified sometimes results in them failing to notice object modifications that occur in other contexts (i.e., a postfix or prefix operation appearing in the right operand of an assignment, or the argument of a function call). It is possible to write the code so that a postfix operator does not occur in the same expression as other operators. Moving the evaluation back before the containing expression removes

1712 full expression

the need for duplicate operations in each arm of an **if/else** (an **else** arm will have to be created if one does not exist) or **switch** statement. For an expression statement the operation can be moved after the containing expression. (If the usage is part of an initializer, the first form is only available in C99, which permits the mixing of declarations and statements.)

```
...i++...
```

becomes the equivalent form:

```
t=i;
i++;
...t...
```

or:

```
...i...
i++;
```

The total cognitive effort needed to comprehend the first equivalent form may be more than the postfix form (the use of `t` has to be comprehended). However, the peak effort may be less (because the operations may have been split into smaller chunks in serial rather than nested form). The total cognitive effort needed to comprehend the second equivalent form may be the same as the original and the peak effort may be less. Are more faults likely to be introduced through miscomprehension, through a visual code scanning strategy that may fail to spot modifications, or through the introduction of a temporary object? There is insufficient evidence to answer this question at the time of this writing. Although in the case of expression statements there is the possibility of a benefit (at a very small initial, typing cost) to moving the modification operation to after the expression. The issue of side effects occurring within expressions containing operators that conditionally evaluate their operands is discussed elsewhere.

&& 1255 second operand

3. *The only operator in a full expression that is not an expression statement.* When the postfix operator is the only operator in a full expression it might be claimed that it will not be overlooked by readers. There is no evidence for, or against, this claim.

Your author has encountered other coding guideline authors, whose primary experience is with non-C like languages, who recommend against the postfix and prefix operators. These recommendations seem to be driven by unfamiliarity. Experience suggests that once the novelty has worn off, these developers become comfortable with and even prefer the shorter C forms.

By their very nature the **++** and **--** operators often occur in expressions used as indices to storage locations. The off-by-one mistake may be the most commonly occurring mistake associated with array subscripting. For some coding guideline authors, this has resulted in guilt by association. There is no evidence to show that the use of these operators leads to more, or less, mistakes being made than if an alternative expression were used.

prefix ++ 1082 incremented

The rationale for preferring prefix operators over postfix operators in some contexts is given elsewhere.

Cg 1046.1

A postfix operator shall not appear in an expression unless it is the only operator in an expression statement.

Semantics

postfix ++ result

The result of the postfix ++ operator is the value of the operand.

Commentary

That is, the value before the object denoted by the operand is incremented.

Common Implementations

Having to return the original value as the result, and to update the original value, requires at least one more register, or storage location, than the prefix form.

Coding Guidelines

An occurrence of the postfix ++ operator may cause the reader to make a cognitive task switch (between evaluating the full expression and evaluating the modification of a storage location). It also requires that two values be temporarily associated with the same object. Having to remember two values requires more memory resources than recalling one. There is also the possibility of interference between the two, closely semantically associated, values (causing one or both of them to be incorrectly recalled). As the previous C sentence discussed, rewriting the code so that a postfix operator is not nested within an expression evaluation may reduce the cognitive resources needed to comprehend that piece of source.

0 cognitive switch

1048 After the result is obtained, the value of the operand is incremented.

Commentary

That is, at the next sequence point the value of the object will be one larger than it was before the evaluation of this operator (assuming no other value is stored into the object before this sequence point is reached, which would result in undefined behavior).

C++

After the result is noted, the value of the object is modified by adding 1 to it, unless the object is of type ***bool****, in which case it is set to* ***true****. [Note: this use is deprecated, see annex D.]*

5.2.6p1

The special case for operands of type **bool** also occurs in C, but a chain of reasoning is required to deduce it.

476 _Bool large enough to store 0 and 1

Common Implementations

The two values created by the postfix operators (the value before and the value after) are likely to simultaneously require two registers, to hold them. The extent to which the greater call on available processor resources, compared to the corresponding prefix operators, results in less optimal generated machine code will depend on the context in which the operator occurs (i.e., a change to the algorithm to use a different operator may affect the quality of the machine code generated for other operators).

1049 (That is, the value 1 of the appropriate type is added to it.)

Commentary

If USHRT_MAX has the same value as INT_MAX, then the following increment would have undefined behavior, if the value 1 used has type **int**. However, it is defined if the value 1 used has type **unsigned int** (it is assumed that the phrase *appropriate type* is to be interpreted as a type for which the behavior is defined).

```
1  unsigned short u = USHRT_MAX;
2
3  u++;
```

1050 See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers.

postfix operators see also

Commentary

addition 1154 operand types compound 1310 assignment constraints additive 1162 operators semantics compound 1312 assignment semantics prefix op- 1085 erators see also

The implication is that these constraints and semantics also apply to the postfix ++ operator. The same references are given for the postfix operators.

C++

5.2.6p1

The C++ Standard provides a reference, but no explicit wording that the conditions described in the cited clauses also apply to this operator.

See also 5.7 and 5.17.

The side effect of updating the stored value of the operand shall occur between the previous and the next sequence point. 1051

Commentary

sequence 187 points

This is a requirement on the implementation. It is a special case of a requirement given elsewhere. The postfix ++ operator is treated the same as any other operator that modifies an object with regard to updating the stored value. It is possible that the ordering of sequence points, during the evaluation of a full expression, is nonunique. It is also possible that the operand may be modified more than once between two sequence points, causing undefined behavior.

sequence 187 points

C++

sequence 187 points

The C++ Standard does not explicitly specify this special case of a more general requirement.

Common Implementations

assignment- 1288 expression syntax

Like the implementation of the assignment operators, there may be advantages to delaying the store operation (keeping the new value in a register), perhaps because subsequent operations may modify it again.

Coding Guidelines

sequence 187.1 points all orderings give same value

The applicable guideline recommendation is discussed elsewhere.

---

postfix -- analogous to ++

The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it). 1052

Commentary

postfix ++ 1047 result

The effects of the -- operator are not analogous to the postfix ++ operator when the operand has type **_Bool** (otherwise the same Commentary and Coding guideline issues also apply). The -- operator always inverts the truth-value of its operand.

```
1  #include <stdio.h>
2
3  void f(_Bool p)
4  {
5  _Bool q = p;
6
7  q--; q++;
8  p++; p--;
9
10 if (p == q)
11    printf("This implementation is not conforming\n");
12 }
```

C++

5.2.6p2

. . . except that the operand shall not be of type **bool**.

A C source file containing an instance of the postfix `--` operator applied to an operand having type **_Bool** is likely to result in a C++ translator issuing a diagnostic.

1053 **Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.2.5 Compound literals

C90

Support for compound literals is new in C99.

C++

Compound literals are new in C99 and are not available in C++.

Usage

No usage information is provided on compound literals because very little existing source code contains any use of them.

Constraints

1054 The type name shall specify an object type or an array of unknown size, but not a variable length array type.

Commentary

The type of the compound literal is deduced from the type name, not from the parenthesized list of expressions. Note that here *type name* refers to a syntactic rule.

1624 abstract declarator syntax

The number of elements in the compound literal are required to be known at translation time. This simplifies the handling of their storage requirements.

Compound literals are not usually thought of in terms of scalar types. In those cases where the address of an object is needed simply to fill an argument slot, using an unnamed object may be simpler than defining a dummy variable.

```
1  extern void glob(int, int, int *);
2
3  void f(void)
4  {
5  int loc = (int){9}; /* Surprising?  Unusual?  Machine generated? */
6  double _Complex C = (double _Complex){2.0 + I * 3.0};
7
8  glob(1, 2, &((int){0}));
9  }
```

Other Languages

Some languages do not require a type name to be given. The type of the parenthesized list of expressions is deduced from the context in which it occurs.

Example

```
1  extern int n;
2  struct incomplete_type;
3
4  void f(void)
5  {
6  typedef int a_n[n];
7  typedef int a_in[];
8
```

```
(a_n){1, 2, 3}; /* Constraint violation. */

/*
 * Number of elements in array deduced from number of
 * expressions in the parentheses; same as for initializers.
 */
(a_in){4, 5, 6, 7};

(void){1.3, 4};                   /* Constraint violation. */
(struct incomplete_type){'a', 3}; /* Constraint violation. */
}
```

initializer value not contained in unnamed object

No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal. 1055

Commentary

initializer 1642 value not contained in object

This is the compound literal equivalent of a constraint on initializers for object definitions. The object referred to here is the object being initialized, not other objects declared within the translation unit (or even unnamed object). For instance:

```
void f(int p)
{
struct s_r {
            int mem_1,
                mem_2;
            int *mem_3;
            };

(int []){ p = 1, /* Also provide a value for p. */
              2};

(struct s_r){1, 7,
             &(([2]){5, 6})}; /* Provide a value for an anonymous object. */
}
```

Example

```
struct TAG {
            int mem;
            };

void f(void)
{
/*
 * More expressions that do not have elements in the array.
 */
(int [3]){0, 1, 2, 3}; /* Constraint violation. */
(int [3]){0, [4] = 1}; /* Constraint violation. */
(struct TAG){5, 6};    /* Constraint violation. */
}
```

If the compound literal occurs outside the body of a function, the initializer list shall consist of constant expressions. 1056

Commentary

All objects defined outside the body of a function have static storage duration. The storage for such objects is initialized before program startup, so can only consist of constant expressions. This constraint only differs from an equivalent one for initializers by being framed in terms of "occurring outside the body of a function" rather than "an object that has static storage duration."

1065 compound literal
151 static storage duration
455 initialized before program startup
1644 initializer
static storage duration object

Semantics

1057 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*.

compound literal

Commentary

This defines the term *compound literal*. A compound literal differs from an initializer list in that it can occur outside of an object definition. Because their need be no associated type definition, a type name must be specified (for initializers the type is obtained from the type of the object being initialized).

1641 initialization
syntax

Other Languages

A form of compound literals are supported in some languages (e.g., Ada, Algol 68, CHILL, and Extended Pascal). These languages do not always require a type name to be given. The type of the parenthesized list of expressions is deduced from the context in which it occurs.

Coding Guidelines

From the coding guideline point of view, the use of compound literals appears fraught with potential pitfalls, including the use of the term *compound literal* which suggests a literal value, not an unnamed object. However, this construct is new in C99 and there is not yet sufficient experience in their use to know if any specific guideline recommendations might apply to them.

1066 compound literal
inside function body
1061 compound literal
is lvalue

1058 It provides an unnamed object whose value is given by the initializer list.[81)]

compound literal
unnamed object

Commentary

The difference between this kind of unnamed object and that created by a call to a memory allocation function (e.g., `malloc`) is that its definition includes a type and it has a storage duration other than allocated (i.e., either static or automatic).

Other Languages

Some languages treat their equivalent of compound literals as just that, a literal. For instance, like other literals, it is not possible to take their address.

Common Implementations

In those cases where a translator can deduce that storage need not be allocated for the unnamed object, the as-if rule can be used, and it need not allocate any storage. This situation is likely to occur for compound literals because, unless their address is taken (explicitly using the address-of operator, or in the case of an array type implicit conversion to pointer type), they are only assigned a value at one location in the source code. At their point of definition, and use, a translator can generate machine code that operates on their constituent values directly rather than copying them to an unnamed object and operating on that.

Coding Guidelines

Guideline recommendations applicable to the unnamed object are the same as those that apply to objects having the same storage duration. For instance, the guideline recommendation dealing with assigning the address of objects to pointers.

1088.1 object
address assigned

Example

The following example not only requires that storage be allocated for the unnamed object created by the compound literal, but that the value it contains be reset on every iteration of the loop.

```
1  struct s_r {
2             int mem;
3             };
4
5  extern void glob(struct s_r *);
6
7  void f(void)
8  {
9  struct s_r *p_s_r;
10
11 while (p_s_r->mem != 10)
12    {
13    glob(p = &((struct s_r){1});
14 /*
15  * Instead of writing the above we could have written:
16  *    struct s_r unnamed_s_r = {1};
17  *    glob (p_s_r = &unnamed_s_r);
18  * which assigns 1 to the member on every iteration, as
19  * part of the process of defining the object.
20  */
21    p_s_r->mem++; /* Increment value held by unnamed object. */
22    }
23 }
```

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.8, and the type of the compound literal is that of the completed array type. 1059

Commentary

array of unknown size initialized 1683

This behavior is discussed elsewhere.

Coding Guidelines

array incomplete type 1573

The some of the issues involved in declaring arrays having an unknown size are discussed elsewhere.

Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. 1060

Commentary

Presumably this is the declared type of the unnamed object initialized by the initializer list and therefore also its effective type.

effective type 948

---

compound literal is lvalue

In either case, the result is an lvalue. 1061

Commentary

lvalue 721

lvalue converted to value 725

While the specification for a compound literal meets the requirements needed to be an lvalue, wording elsewhere might be read to imply that the result is not an lvalue. This specification clarifies the behavior.

Other Languages

rvalue 736

Some languages consider, their equivalent of, compound literals to be just that, literals. For such languages the result is an rvalue.

---

footnote 81

81) Note that this differs from a cast expression. 1062

Commentary

A cast operator takes a single scalar value (if necessary any lvalue is converted to its value) as its operand and returns a value as its result.

Coding Guidelines

Developers are unlikely to write expressions, such as `(int){1}`, when `(int)1` had been intended (on standard US PC-compatible keyboards the pair of characters *({* and the pair *) }* appear on four different keys). Such usage may occur through the use of parameterized macros. However, at the time of this writing there is insufficient experience with use of this new language construct to know whether any guideline recommendation is worthwhile.

Example

The following all assign a value to `loc`. The first two assignments involve an lvalue to value conversion. In the second two assignments the operand being assigned is already a value.

```
1   extern int glob = 1;
2
3   void f(void)
4   {
5   int loc;
6
7   loc=glob;
8   loc=(int){1};
9
10  loc=2;
11  loc=(int)2;
12  }
```

1063 For example, a cast specifies a conversion to scalar types or **`void`** only, and the result of a cast expression is not an lvalue.

Commentary

These are restrictions on the types and operands of such an expression and one property of its result.

1134 cast
scalar or void type
1131 footnote 85

Example

```
1   &(int)x;   /* Constraint violation. */
2   &(int){x}; /* Address of an unnamed object containing the current value of x. */
```

1064 The value of the compound literal is that of an unnamed object initialized by the initializer list.

Commentary

The distinction between a compound literal acting as if the initializer list was its value, and an unnamed object (initialized with values from the initializer list) being its value, is only apparent when the address-of operator is applied to it. The creation of an unnamed object does not mean that locally allocated storage is a factor in this distinction. Implementations of languages where compound literals are defined to be literals sometimes use locally allocated temporary storage to hold their values. C implementations may find they can optimize away allocation of any actual unnamed storage.

Common Implementations

If a compound literal occurs in a context where its value is required (e.g., assignment) there are obvious opportunities for implementations to use the values of the initializer list directly. C99 is still too new to know whether most implementations will make use of this optimization.

Coding Guidelines

The distinction between the value of a compound literal being an unnamed object and being the values of the initializer list could be viewed as an unnecessary complication that is not worth educating a developer about. Until more experience has been gained with the kinds of mistakes developers make with compound literals, it is not possible to recommend any guidelines.

Example

```
#include <string.h>

struct TAG {
            int mem_1;
            float mem_2;
           };

struct TAG o_s1 = (struct TAG){1, 2.3};

void f(void)
{
memcpy(&o_s1, &(struct TAG){4, 5.6}, sizeof(struct TAG));
}
```

---

compound literal outside function body

If the compound literal occurs outside the body of a function, the object has static storage duration; 1065

Commentary

storage 448 duration object

This specification is consistent with how other object declarations, outside of function bodies, behave. The storage duration of a compound literal is based on the context in which it occurs, not whether its initializer list consists of constant expressions.

```
struct s_r {
           int mem;
           };

static struct s_r glob =              {4};
static struct s_r bolg =  (struct s_r){4}; /* Constraint violation. */
static struct s_r *p_g = &(struct s_r){4};

void f(void)
{
static struct s_r  loc =              {4};
static struct s_r  col =  (struct s_r){4}; /* Constraint violation. */
static struct s_r *p_l = &(struct s_r){4}; /* Constraint violation. */
}
```

Other Languages

The storage duration specified by other languages, which support some form of compound literal, varies. Some allow the developer to choose (e.g., Algol 68), others require them to be dynamically allocated (e.g., Ada), while in others (e.g., Fortran and Pascal) the issue is irrelevant because it is not possible to obtain their address.

---

compound literal inside function body

otherwise, it has automatic storage duration associated with the enclosing block. 1066

Commentary

A parallel can be drawn between an object definition that includes an initializer and a compound literal (that is the definition of an unnamed object). The lifetime of the associated objects starts when the block that

458 object lifetime from entry to exit of block
462 initialization performed every time declaration reached
1711 object initializer evaluated when
1026 function call recursive
1078 EXAMPLE compound literal single object

contains their definition is entered. However, the objects are not assigned their initial value, if any, until the declaration is encountered during program execution.

The unnamed object associated with a compound literal is initialized each time the statement that contains it is encountered during program execution. Previous invocations, which may have modified the value of the unnamed object, or nested invocations in a recursive call, do not affect the value of the newly created object. Storage for the unnamed object is created on block entry. Executing a statement containing a compound literal does not cause any new storage to be allocated. Recursive calls to a function containing a compound literal will cause different storage to be allocated, for the unnamed object, for each nested call.

```
1   struct foo {
2               struct foo *next;
3               int i;
4               };
5
6   void WG14_N759(void)
7   {
8   struct foo *p,
9               *q;
10  /*
11   * The following loop ...
12   */
13  p = NULL;
14  for (int j = 0; j < 10; j++)
15     {
16     q = &((struct foo){ .next = p, .i = j });
17     p = q;
18     }
19  /*
20   * ... is equivalent to the loop below.
21   */
22  p = NULL;
23  for (int j = 0; j < 10; j++)
24     {
25     struct foo T;
26
27     T.next = p;
28     T.i = j;
29     q = &T;
30     p = q;
31     }
32  }
```

Common Implementations

To what extent is it worth trying to optimize compound literals made up of a list of constant expressions; for instance, by detecting those that are never modified, or by placing them in a static region of storage that can be copied from or pointed at? The answer to these and many other optimization issues relating to compound literals will have to wait until translator vendors get a feel for how their customers use this new, to C, construct.

Coding Guidelines

Parallels can be drawn between the unnamed object associated with a compound literal and the temporaries created in C++. Experience has shown that C++ developers sometimes assume that the lifetime of a temporary is greater than it is required to be by that languages standard. Based on this experience it is to be expected that developers using C might make similar mistakes with the lifetime of the unnamed object associated with a compound literal. Only time will tell whether these mistakes will be sufficiently common, or serious, that the benefits of being able to apply the address-of operator to a compound literal (the operator that needs to be used to extend the range of statements over which an unnamed object can be accessed) are outweighed by the probably cost of faults.

object address assigned 1088.1

The guideline recommendation dealing with assigning the address of an object to a pointer object, whose lifetime is greater than that of the addressed object, is applicable here.

```
#include <stdlib.h>

extern int glob;
struct s_r {
           int mem;
           };

void f(void)
{
struct s_r *p_s_r;

if (glob == 0)
   {
   p_s_r = &((struct s_r){1});
   }
else
   {
   p_s_r = &((struct s_r){2});
   }
/* The value of p_s_r is indeterminate here. */

/*
 * The iteration-statements all enclose their associated bodies in
 * a block.  The effect of this block is to start and terminate
 * the lifetime of the contained compound literal.
 */
p_s_r=NULL;
while (glob < 10)
   {
   /*
    * In the following test the value of p_s_r is indeterminate
    * on the second and subsequent iterations of the loop.
    */
   if (p_s_r == NULL)
      ;
   p_s_r = &((struct s_r){1});
   }
}
```

All the semantic rules and constraints for initializer lists in 6.7.8 are applicable to compound literals.[82)] 1067

Commentary

They are the same except

- initializer lists don't create objects, they are simply a list of values with which to initialize an object; and
- the type is deduced from the object being initialized, not a type name.

Coding Guidelines

initialization syntax 1641

Many of the coding guideline issues discussed for initializers also apply to compound literals.

String literals, and compound literals with const-qualified types, need not designate distinct objects.[83)] 1068

string literal distinct object
compound literal distinct object

Commentary

A strictly conforming program can deduce if an implementation uses the same object for two string literals, or compound literals, by performing an equality comparison on their addresses (an infinite number of comparisons would be needed to deduce whether an implementation always used distinct objects). This permission for string literals is also specified elsewhere.

1076 EXAMPLE string literals shared
908 string literal distinct array

The only way a const-qualified object can be modified is by casting a pointer to it to a non-const-qualified pointer. Such usage results in undefined behavior. The undefined behavior, if the pointer was used to modify such an unnamed object that was not distinct, could also modify the values of other compound literal object values.

746 pointer converting qualified/unqualified

Other Languages

Most languages do not consider any kind of literal to be modifiable, so whether they share the same storage locations is not an issue.

Common Implementations

The extent to which developers will use compound literals having a const-qualified type, for which storage is allocated and whose values form a sharable subset with another compound literal, remains to be seen. Without such usage it is unlikely that implementors of optimizers will specifically look for savings in this area, although they may come about as a consequence of optimizations not specifically aimed at compound literals.

Example

In the following there is an opportunity to overlay the two unnamed objects containing zero values.

```
1   const int *p1 = (const int [99]){0};
2   const int *p2 = (const int [20]){0};
```

1069 EXAMPLE 1 The file scope definition

```
int *p = (int []){2, 4};
```

initializes **p** to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

Commentary

This usage, rather than the more obvious `int p[] = {2, 4};`, can arise because the initialization value is derived through macro replacement. The same macro replacement is used in noninitialization contexts.

1070 EXAMPLE 2 In contrast, in

```
void f(void)
{
        int *p;
        /* ... */
        p = (int [2]){*p};
        /* ... */
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by **p** and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

Commentary

The assignment of values to the unnamed object occurs before the value of the right operand is assigned to p.

Example

The above example is not the same as declaring p to be an array.

```
1  void f(void)
2  {
3  int p[2]; /* Storage for p is created by its definition. */
4
5  /*
6   * Cannot assign new object to p, can only change existing values.
7   */
8  p[1]=0;
9  }
```

EXAMPLE 3 Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order: 1071

```
drawline((struct point){.x=1, .y=1},
         (struct point){.x=3, .y=4});
```

Or, if **drawline** instead expected pointers to **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
         &(struct point){.x=3, .y=4});
```

Commentary

This usage removes the need to create a temporary in the calling function. The arguments are passed by value, like any other structure argument.

EXAMPLE 4 A read-only compound literal can be specified through constructions like: 1072

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

Commentary

An implementation may choose to place the contents of this compound literal in read-only memory, but it is not required to do so. The term *read-only* is something of a misnomer, since it is possible to cast its address to a non-const-qualified type and assign to the pointed-to object. (The behavior is undefined, but unless the values are held in a kind of storage that cannot be modified, they are likely to be modified.)

Other Languages

Some languages support a proper read-only qualifier.

Common Implementations

On some freestanding implementations this compound literal might be held in ROM.

---

footnote 82

82) For example, subobjects without explicit initializers are initialized to zero. 1073

Commentary

initializer 1682 fewer in list than members

This behavior reduces the volume of the visible source code when the object type includes large numbers of members or elements.

Coding Guidelines

initialization 1641 syntax

Some of the readability issues applicable to statements have different priorities than those for declarations. These are discussed elsewhere.

---

footnote 83

83) This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations. 1074

Commentary

The need to discuss an implementation's ability to share storage for string literals occurs because it is possible to detect such sharing in a conforming program (e.g., by comparing two pointers assigned the addresses of two distinct, in the visible source code, string literals). The C Committee choose to permit this implementation behavior. (There were existing implementations, when the C90 Standard was being drafted, that shared storage.)

1075 EXAMPLE 5 The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

Commentary

In all three cases, a pointer to the start of storage is returned and the first 16 bytes of the storage allocated will have the same set of values. If all three expressions occurred in the same source file, the first and third could share the same storage even though their storage durations were different. Developers who see a potential storage saving in using a compound literal instead of a string literal (the storage for one only need be allocated during the lifetime of its enclosing block) also need to consider potential differences in the number of machine code instructions that will be generated. Overall, there may be no savings.

1076 EXAMPLE string literals shared

1076 EXAMPLE 6 Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

EXAMPLE string literals shared

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

Commentary

In this example pointers to the first element of the compound literal and a string literal are being compared for equality. Permission to share the storage allocated for a compound literal only applies to those having a const-qualified type (there is no such restriction on string literals).

1068 compound literal distinct object
908 string literal distinct array

Coding Guidelines

Comparing string using an equality operator, rather than a call to the `strcmp` library function is a common beginner mistake. Training is the obvious solution.

Usage

In the visible source of the `.c` files 0.1% of string literals appeared as the operand of the equality operator (representing 0.3% of the occurrences of this operator).

1077 EXAMPLE 7 Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object **endless_zeros** below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

Commentary

A modification using pointer types, and an additional assignment, creates a circularly linked list that uses the storage of the unnamed object:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list *endless_zeros = &(struct int_list){0, 0};

endless_zeros->cdr=endless_zeros; /* Let's follow ourselves. */
```

The following statement would not have achieved the same result:

```
endless_zeros = &(struct int_list){0, endless_zeros};
```

because the second compound literal would occupy a distinct object, different from the first. The value of `endless_zeros` in the second compound literal would be pointing at the unnamed object allocated for the first compound literal.

Other Languages

Algol 68 supports the creation of circularly linked objects (see the Other Languages subsection in the following C sentence).

EXAMPLE compound literal single object

EXAMPLE 8 Each compound literal creates only a single object in a given scope: 1078

```
struct s { int i; };

int f (void)
{
        struct s *p = 0, *q;
        int j = 0;

again:
        q = p, p = &((struct s){ j++ });
        if (j < 2) goto again;

        return p == q && q->i == 1;
}
```

The function **f()** always returns the value 1.

Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around **p** would have an indeterminate value, which would result in undefined behavior.

Commentary

Specifying that a single object is created helps prevent innocent-looking code consuming large amounts of storage (e.g., use of a compound literal in a loop).

Other Languages

In Algol 68 **LOC** creates storage for block scope objects. However, it generates new storage every time it is executed. The following allocates 1,000 objects on the stack.

```
MODE M = STRUCT (REF M next, INT i);
M p;
INT i := 0

again:
    p := LOC M := (p, i);
    i +:= 1;
    IF i < 1000 THEN
       GO TO again
    FI;
```

1079 **Forward references:** type names (6.7.6), initialization (6.7.8).

6.5.3 Unary operators

unary-expression syntax

1080

```
unary-expression:
                postfix-expression
                ++ unary-expression
                -- unary-expression
                unary-operator cast-expression
                sizeof unary-expression
                sizeof ( type-name )
unary-operator: one of
                &  *  +  -  ~  !
```

Commentary

Note that the operand of unary-operator is a *cast-expression*, not a *unary-expression*. A unary operator usually refers to an operator that takes a single argument. Technically all of the operators listed here, plus the postfix increment and decrement operators, could be considered as being unary operators.

1133 cast-expression syntax

Unary plus was adopted by the C89 Committee from several implementations, for symmetry with unary minus. Rationale

Other Languages

Some languages (i.e., Ada and Pascal) specify the unary operators to have lower precedence than the multiplicative operators; for instance, `-x/y` is equivalent to `-(x/y)` in Ada, but `(-x)/y` in C. Most languages call all operators that take a single-operand unary operators.

1143 multiplicative-expression syntax

Languages that support the unary `+` operator include Ada, Fortran, and Pascal. Some languages use the keyword **NOT** rather than `!`. In the case of Cobol this keyword can also appear to the left of an operator, indicating negation of the operator (i.e., **NOT** < meaning not less than).

Coding Guidelines

Coding guidelines need to be careful in their use of the term *unary operator*. Its meaning, as developers understand it, may be different from its actual definition in C. The operators in a *unary-expression* occur to the left of the operand. The only situation where a developer's incorrect assumption about precedence relationships might lead to a difference between predicted and actual behavior is when a postfix operator occurs immediately to the right of the *unary-expression*.

Dev 943.1

Except when *sizeof (type-name)* is immediately followed visually by a token having the lexical form of an additive operator, if a *unary-expression* is not immediately followed by a postfix operator it need not be parenthesized.

Although the expression `sizeof (int)-1` may not occur in the visible source code, it could easily occur as the result of macro replacement of the operand of the **sizeof** operator. This is one of the reasons behind the guideline recommendation specifying the parenthesizing of macro bodies (without parentheses the expression is equivalent to `(sizeof(int))-1`).

1931.2 macro definition expression

Example

```
1  struct s {
2            int x;
3            };
```

```
4   struct s *a;
5   int x;
6
7   void f(void)
8   {
9   x<-a->x;
10  x<--a->x;
11  x<- --a->x;
12  x<- - --a->x;
13
14  sizeof(long)-3; /* Could be mistaken for sizeof a cast-expression. */
15  (sizeof(long))-3;
16  sizeof((long)-3);
17  }
```

Usage

postfix-985 expression syntax

See the Usage section of *postfix-expression* for `++` and `--` digraph percentages.

Table 1080.1: Common token pairs involving **sizeof**, *unary-operator*, prefix `++`, or prefix `--` (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
`!` defined	2.0	16.7	`!` `(`	14.5	0.5
`*v --v`	0.3	7.8	`-v` identifier	30.2	0.4
`-v` *floating-constant*	0.3	6.7	`*v (`	9.0	0.4
`*v ++v`	0.5	6.3	`~` *integer-constant*	20.1	0.2
`! --v`	0.2	4.8	`++v` identifier	97.3	0.1
`-v` *integer-constant*	69.0	4.1	`~` identifier	56.3	0.1
`&v` identifier	96.1	1.9	`~ (`	23.4	0.1
sizeof `(`	97.5	1.8	`+v` *integer-constant*	49.0	0.0
`*v` identifier	86.8	1.0	`--v` identifier	97.1	0.0
`!` identifier	81.9	0.8			

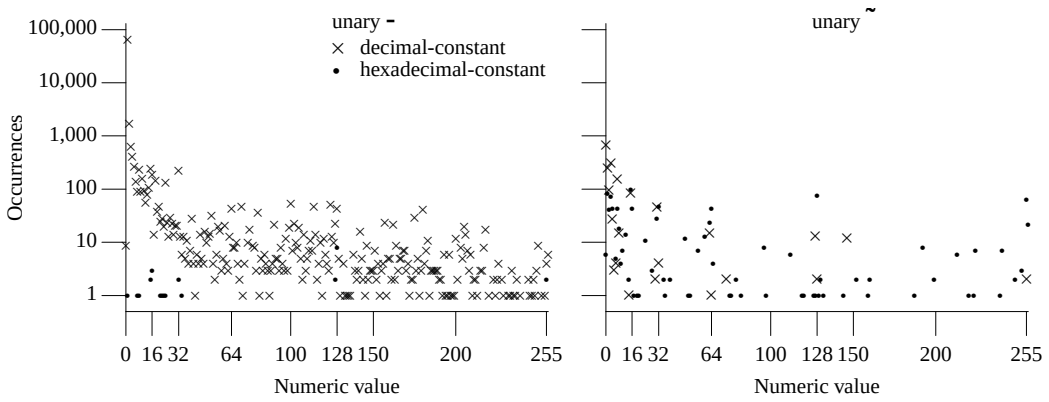


Figure 1080.1: Number of *integer-constant*s having a given value appearing as the operand of the unary minus and unary `~` operators. Based on the visible form of the `.c` files.

Table 1080.2: Occurrence of the *unary-operator*s, prefix ++, and prefix -- having particular operand types (as a percentage of all occurrences of the particular operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Operator	Type	%	Operator	Type	%	Operator	Type	%
-v	_int	96.0	~	unsigned long	6.8	!	_long	2.7
*v	ptr-to	95.3	&v	int	6.2	~	unsigned char	2.5
+v	_int	72.2	~	unsigned int	6.0	&v	unsigned char	2.4
--v	int	54.7	+v	unsigned long	5.6	!	unsigned long	2.1
!	int	50.0	+v	long	5.6	~	long	2.0
~	_int	49.3	+v	float	5.6	++v	unsigned char	1.9
&v	other-types	45.1	!	other-types	5.6	~	_unsigned long	1.7
++v	int	43.8	++v	unsigned long	5.2	~	_unsigned int	1.7
++v	ptr-to	33.3	&v	struct *	4.9	!	unsigned char	1.6
~	int	28.5	--v	unsigned long	4.7	~	other-types	1.6
--v	unsigned int	22.1	!	unsigned int	4.7	-v	_double	1.4
!	ptr-to	20.1	*v	fnptr-to	4.1	-v	other-types	1.3
--v	ptr-to	14.6	&v	unsigned long	4.0	++v	long	1.2
&v	struct	13.9	--v	other-types	4.0	-v	int	1.2
&v	char	13.1	&v	long	3.4	!	_int	1.2
++v	unsigned int	12.6	&v	unsigned int	3.0	++v	unsigned short	1.1
+v	int	11.1	&v	unsigned short	2.9	&v	char *	1.1
!	char	9.2	!	enum	2.9			

6.5.3.1 Prefix increment and decrement operators

Constraints

1081 The operand of the prefix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue.

postfix operator operand

Commentary

This constraint mirrors that for the postfix forms of these operators.

1081 postfix operator operand

C++

The use of an operand of type **bool** with the prefix ++ operator is deprecated (5.3.2p1); there is no corresponding entry in Annex D, but the proposed response to C++ DR #145 inserted one. In the case of the decrement operator:

> *The operand shall not be of type* ***bool****.* 5.3.2p1

A C source file containing an instance of the prefix -- operator applied to an operand having type **_Bool** is likely to result in a C++ translator issuing a diagnostic.

Coding Guidelines

Enumerated types are usually thought about in symbolic rather than arithmetic terms. The increment and decrement operators can also be given a symbolic interpretation. They are sometimes thought about in terms of moving on to the next symbolic name in a list. This *move to next* operation relies on the enumeration constants being represented by successive numeric values. While this usage is making use of representation information, there is often a need to step through a series of symbolic names (and C provides no other built-in mechanism), for instance, iterating over the named constants defined by an enumerated type.

822 symbolic name
517 enumeration set of named constants
1199 relational operators real operands

Dev 569.1 The operand of a prefix increment or decrement operator may have an enumerated type, provided the enumeration constants defined by that type have successive numeric values.

Semantics

prefix ++ incremented

The value of the operand of the prefix ++ operator is incremented. 1082

Commentary

postfix ++ result 1047

The ordering of this and the following C sentence is the reverse of that specified for the postfix ++ operator.

Common Implementations

The implementation of this operator is usually very straight-forward. A value is loaded into a register, incremented, and then stored back into the original object, leaving the result in the register. Some CISC processors contain instructions that increment the contents of storage directly. Processors that have a stack-based architecture either need to contain store instructions that leave the value on the stack, or be willing to pay the penalty of another load from storage.

Coding Guidelines

Translators have now progressed to the point where the optimizations many of them perform are much more sophisticated than those needed to detect the more verbose sequence of operations equivalent to the prefix ++ operator. The writers of optimizers study existing source code to find out what constructs occur frequently (they don't want to waste time and money implementing optimizations for constructs that rarely occur). However, in existing code it is rare to see an object being incremented (or decremented) without one of these operators being used. Consequently optimizers are unlikely to attempt to transform the C source `i=i+1` into `++i` (which they might have to do for say Pascal, which has no increment operators requiring optimizers to analyze an expression looking for operations that are effectively increment object). So the assertion that `++i` can be written as `i=i+1` and that it will be optimized by the translator is not guaranteed, even for a highly optimizing translator. However, this is rarely an important issue anyway; the difference in quality of generated machine code rarely has any impact on program performance.

From the coding guidelines perspective, uses of these operators can be grouped into three categories:

1. The only operator in an expression statement. In this context the result returned by the operation is ignored. The statement simply increments/decrements its operand. Use of the prefix, rather than the postfix, form does not follow the pattern seen at the start of most visible source code statement lines—an identifier followed by an operator (see Figure 940.2). A reader's scanning of the source looking for objects that are modified will be disrupted by the initial operator. For this reason, use of the postfix form is recommended.

   postfix operator constraint 1046

2. One of the operators in a full expression that contains other operators. It is possible to write the code so that a prefix operator does not occur in the same expression as other operators. The evaluation can be moved back before the containing expression (see the postfix operators for a fuller discussion of this point).

   full expression 1712

   postfix operator constraint 1046

   ```
   ...++i...
   ```

   becomes the equivalent form:

   ```
   i++;
   ...i...
   ```

   The total cognitive effort needed to comprehend the equivalent form may be less than the prefix form, and the peak effort is likely to be less (because the operations may have been split into smaller chunks in serial rather than nested form).

3. The third point is the same as for the postfix operators.

   postfix operator constraint 1046

Cg 1082.1

The prefix operators shall not appear in an expression statement.

1083 The result is the new value of the operand after incrementation.

prefix ++ result

Other Languages

Pascal contains the **succ** operator. This returns the successor value (i.e., it adds one to its operand), but it does not modify the value of an object appearing as its operand.

1084 The expression `++E` is equivalent to `(E+=1)`.

Commentary

The expression ++E need not be equivalent to E=E+1 (e.g., the expression E may contain a side effect).

C++

C++ lists an exception (5.3.2p1) for the case when E has type **bool**. This is needed because C++ does not define its boolean type in the same way as C. The behavior of this operator on operands is defined as a special case in C++. The final result is the same as in C.

476 _Bool large enough to store 0 and 1

1085 See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

prefix operators see also

Commentary

The same references are given for the postfix operators.

1050 postfix operators see also

C++

[Note: see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions.] 5.3.2p1

There is no mention that the conditions described in these clauses also apply to this operator.

1086 The prefix -- operator is analogous to the prefix ++ operator, except that the value of the operand is decremented.

Commentary

The same Commentary and Coding Guidelines' issues also apply. See the discussion elsewhere for cases where the affects are not analogous.

1082 prefix ++ incremented
1052 postfix -- analogous to ++

C++

The prefix **--** operator is not analogous to the prefix **++** operator in that its operand may not have type **bool**.

Other Languages

Pascal contains the **pred** reserved identifier. This returns the predecessor value, but does not modify the value of its operand.

Coding Guidelines

The guideline recommendation for the prefix **++** operator has been worded to apply to either operator.

1082.1 prefix in expression statement

1087 **Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.3.2 Address and indirection operators

Constraints

1088 The operand of the unary **&** operator shall be either a function designator, the result of a **[]** or unary * operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier.

unary & operand constraints

Commentary

bit-field 1410 packed into

byte 53 addressable unit

register 1369 storage-class

Bit-fields are permitted (intended even) to occupy part of a storage unit. Requiring bit addressing could be a huge burden on implementations. Very few processors support bit addressing and C is based on the byte being the basic unit of addressability.

The **register** storage-class specifier is only a hint to the translator. Taking the address of an object could effectively prevent a translator from keeping its value in a register. A harmless consequence, but the C Committee decided to make it a constraint violation.

C90

The words:

> *. . . , the result of a* **[ ]** *or unary* * *operator,*

are new in C99 and were added to cover the following case:

```
int a[10];

for (int *p = &a[0]; p < &a[10]; p++)
    /* ... */
```

where C90 requires the operand to refer to an object. The expression `a+10` exists, but does not refer to an object. In C90 the expression `&a[10]` is undefined behavior, while C99 defines the behavior.

C++

Like C90 the C++ Standard does not say anything explicit about the result of a **[]** or unary * operator. The C++ Standard does not explicitly exclude objects declared with the **register** storage-class specifier appearing as operands of the unary **&** operator. In fact, there is wording suggesting that such a usage is permitted:

7.1.1p3 > *A* **register** *specifier has the same semantics as an* **auto** *specifier together with a hint to the implementation that the object so declared will be heavily used. [Note: the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. —end note]*

Source developed using a C++ translator may contain occurrences of the unary **&** operator applied to an operand declared with the **register** storage-class specifier, which will cause a constraint violation if processed by a C translator.

```
void f(void)
{
register int a[10]; /* undefined behavior */
                    // well-formed

&a[1]  /* constraint violation */
       // well-formed
     ;
}
```

Other Languages

Many languages that support pointers have no address operator (e.g., Pascal and Java, which has references, not pointers). In these languages, pointers can only point at objects returned by the memory-allocation functions. The address-of operator was introduced in Ada 95 (it was not in available in Ada 83). Many languages do not allow the address of a function to be taken.

Coding Guidelines

In itself, use of the address-of operator is relatively harmless. The problems occur subsequently when the value returned is used to access storage. The following are three, coding guideline related, consequences of being able to take the address of an object:

- It provides another mechanism for accessing the individual bytes of an object representation (a pointer to an object can be cast to a pointer to character type, enabling the individual bytes of an object representation to be accessed). 761 pointer converted to pointer to character
- It is an alias for the object having that address.
- It provides a mechanism for accessing the storage allocated to an object after the lifetime of that object has terminated.

Assigning the address of an object potentially increases the scope over which that object can be accessed. When is it necessary to increase the scope of an object? What are the costs/benefits of referring to an object using its address rather than its name? (If a larger scope is needed, could an objects definition be moved to a scope where it is visible to all source code statements that need to refer to it?)

The parameter-passing mechanism in C is pass by value. What is often known as *pass by reference* is achieved, in C, by explicitly passing the address of an object. Different calls to a function having pass-by-reference arguments can involve different objects in different calls. Passing arguments, by reference, to functions is not a necessity; it is possible to pass information into and out of functions using file scope objects. 1004 function call preparing for

Assigning the address of an object creates an alias for that object. It then becomes possible to access the same object in more than one way. The use of aliases creates technical problems for translators (the behavior implied by the use of the **`restrict`** keyword was introduced into C99 to help get around this problem) and can require developers to use additional cognitive resources (they need to keep track of aliased objects). 1491 restrict intended use

A classification often implicitly made by developers is to categorize objects based on how they are accessed, the two categories being those accessed by the name they were declared with and those accessed via pointers. A consequence of using this classification is that developers overlook the possibility, within a sequence of statements, of a particular object being modified via both methods. When readers are aware of an object having two modes of reference (a name and a pointer dereference) is additional cognitive effort needed to comprehend the source? Your author knows of no research in on this subject. These coding guidelines discuss the aliasing issue purely from the oversight point of view (faults being introduced because of lack of information), because there is no known experimental evidence for any cognitive factors.

One way of reducing aliasing issues at the point of object access is to reduce the number of objects whose addresses are taken. Is it possible to specify a set of objects whose addresses should not be taken and what are the costs of having no alternatives for these cases? Is the cost worth the benefit? Restricting the operands of the address operator to be objects having block scope would limit the scope over which aliasing could occur. However, there are situations where the addresses of objects at file scope needs to be used, including:

- An argument to a function could be an object with block scope, or file scope; for instance, the `qsort` function might be called.
- In resource-constrained environments it may be decided not to use dynamic storage allocation. For instance, all of the required storage may be defined at file scope and pointers to objects within this storage used by the program.
- The return from a function call is sometimes a pointer to an object, holding information. It may simplify storage management if this is a pointer to an object at file scope.

The following guideline recommendation ensures that the storage allocated to an object is not accessed once the object's lifetime has terminated.

Cg 1088.1 The address of an object shall not be assigned to another object whose scope is greater than that of the object assigned.

Dev 1088.1 An object defined in block scope, having static storage duration, may have its address assigned to any other object.

function designator converted to type 732 array converted to pointer 729

A function designator can appear as the operand of the address-of operator. However, taking the address of a function is redundant. This issue is discussed elsewhere. Likewise for objects having an array type.

Example

In the following it is not possible to take the address of a or any of its elements.

```
1  register int a[3];
```

In fact this object is virtually useless (the identifier a can appear as the operand to the **sizeof** operator). If allocated memory is not permitted (we know the memory requirements of the following on program startup):

```
1  extern int *p;
2
3  void init(void)
4  {
5  static int p_obj[20];
6
7  p=&p_obj;
8  }
```

This provides pointers to objects, but hides those objects within a block scope. There is no pointer/identifier aliasing problem.

---

unary * operand has pointer type

The operand of the unary * operator shall have pointer type. 1089

Commentary

unary * result type 1098

Depending on the context in which it occurs, there may be restrictions on the pointed-to type (because of the type of the result).

C++

5.3.1p1 *The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type . . .*

C++ does not permit the unary * operator to be applied to an operand having a pointer to **void** type.

```
1  void *g_ptr;
2
3  void f(void)
4  {
5  &*g_ptr; /* DR #012 */
6           // DR #232
7  }
```

Other Languages

In some languages indirection is a postfix operator; for instance, Pascal uses the token ^ as a postfix operator.

Semantics

1090 The unary & operator yields the address of its operand.

unary & operator

Commentary

For operands with static storage duration, the value of the address operator may be a constant (objects having an array type also need to be indexed with a constant expression). There is no requirement that the address of an object be the same between different executions of the same program image (for objects with static storage duration) or different executions of the same function (for objects with automatic storage duration).

1341 address constant

All external function references are resolved during translation phase 8. Any identifier denoting a function definition will have been resolved.

139 translation phase 8

The C99 Standard refers to this as the *address-of* operator.

1014 footnote 79

C90

This sentence is new in C99 and summarizes what the unary **&** operator does.

C++

Like C90, the C++ Standard specifies a pointer to its operand (5.3.1p1). But later on (5.3.1p2) goes on to say: "In particular, the address of an object of type "*cv* T" is "pointer to *cv* T," with the same cv-qualifiers."

Other Languages

Many languages do not contain an address-of operator. Fortran 95 has an address assignment operator, **=>**. The left operand is assigned the address of the right operand.

Common Implementations

Early versions of K&R C treated p=&x as being equivalent to p&=x.[723]

In the case of constant addresses the value used in the program image is often calculated at link-time. For objects with automatic storage duration, their address is usually calculated by adding a known, at translation time, value (the offset of an object within its local storage area) to the value of the frame pointer for that function invocation. Addresses of elements, or members, of objects can be calculated using the base address of the object plus the offset of the corresponding subobject.

Having an object appear as the operand of the address-of operator causes many implementations to play safe and not attempt to perform some optimizations on that object. For instance, without sophisticated pointer analysis, it is not possible to know which object a pointer dereference will access. (Implementations often assume all objects that have had their address taken are possible candidates, others might use information on the pointed-to type to attempt to reduce the set of possible accessed objects.) This often results in no attempt being made to keep the values of such objects in registers.

Implementations' representation of addresses is discussed elsewhere.

540 pointer type describes a

1091 If the operand has type "*type*", the result has type "pointer to *type*".

Commentary

Although developers often refer to the address returned by the address-of operator, C does not have an *address* type.

1092 If the operand is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue.

&*

Commentary

The only effect of the operator pair &* is to remove any lvalueness from the underlying operand. The combination *& returns an lvalue if its operand is an lvalue. This specification is consistent with the behavior of the last operator applied controlling lvalue-ness. This case was added in C99 to cover a number of existing coding idioms; for instance:

1114 footnote 84

1115 *&

```
#include <stddef.h>

void DR_076(void)
{
int *n = NULL;
int *p;

/*
 * The following case is most likely to occur when the
 * expression *n is a macro argument, or body of a macro.
 */
p = &*n;
/* ... */
}
```

C90

The responses to DR #012, DR #076, and DR #106 specified that the above constructs were constraint violations. However, no C90 implementations known to your author diagnosed occurrences of these constructs.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place.

At the time of this writing a response to C++ DR #232 is being drafted (a note from the Oct 2003 WG21 meeting says: "We agreed that the approach in the standard seems okay: `p = 0; *p;` is not inherently an error. An lvalue-to-rvalue conversion would give it undefined behavior.").

```
void DR_232(void)
{
int *loc = 0;

if (&*loc == 0) /* no dereference of a null pointer, defined behavior */
                // probably not a dereference of a null pointer.
   ;

&*loc = 0; /* not an lvalue in C */
           // how should an implementation interpret the phrase must not (5.3.1p1)?
}
```

Common Implementations

Some C90 implementations did not optimize the operator pair &* into a no-op. In these implementations the behavior of the unary * operator was not altered by the subsequent address-of operator. C99 implementations are required to optimize away the operator pair &*.

Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary * that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator. 1093

Commentary

This case was added in C99 to cover a number of coding idioms; for instance:

```
void DR_076(void)
{
int a[10];
int *p;

```

```
6  /*
7   * It is possible to point one past the end of an object.
8   * For instance, we might want to loop over an object, using
9   * this one past the end value.  Given the equivalence that
10  * applies to the subscript operator the operand of & in the
11  * following case is the result of a unary * operator.
12  */
13 p = &a[10];
14
15 for (p = &a[0]; p < &a[10]; p++)
16    /* ... */ ;
17 }
```

C90

This requirement was not explicitly specified in the C90 Standard. It was the subject of a DR #076 that was closed by adding this wording to the C99 Standard.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place. The response to C++ DR #232 may specify the behavior for this case.

Common Implementations

This requirement describes how all known C90 implementations behave.

Coding Guidelines

The expression `&a[index]`, in the visible source code, could imply

- a lack of knowledge of C semantics (why wasn't `a+index` written?),
- that the developer is trying to make the intent explicit, and
- that the developer is adhering to a coding standard that recommends against the use of pointer arithmetic— the authors of such standards often view `(a+index)` as pointer arithmetic, but `a[index]` as an array index (the equivalence between these two forms being lost on them).

989 array subscript identical to

1094 Otherwise, the result is a pointer to the object or function designated by its operand.

Commentary

There is no difference between the use of objects having a pointer type and using the address-of operator. For instance, the result of the address-of operator could be assigned to an object having the appropriate pointer type, and that object used interchangeably with the value assigned to it.

Common Implementations

In most implementations a pointer refers to the actual address of an object or function.

540 pointer type describes a

1095 The unary * operator denotes indirection.

unary * indirection

Commentary

The terms *indirection* and *dereference* are both commonly used by developers.

C++

*The unary * operator performs indirection.*

Other Languages

Some languages (e.g., Pascal and Ada) use the postfix operator `^`. Other languages— Algol 68 and Fortran 95— implicitly perform the indirection operation. In this case, an occurrence of operand, having a pointer type, is dereferenced to return the value of the pointed-to object.

Coding Guidelines

Some coding guideline documents place a maximum limit on the number of simultaneous indirection operators that can be successively applied. The rationale being that deeply nested indirections can be difficult to comprehend. Is there any substance to this claim?

sequential nesting * / sequential 1000 nesting ()

Expressions, such as `***p`, are similar to nested function calls in that they have to be comprehended in a right-to-left order. The issue of nested constructions in natural language is discussed in that earlier C sentence. At the time of this writing there is insufficient experimental evidence to enable a meaningful cost/benefit analysis to be performed and these coding guidelines say nothing more about this issue.

If sequences of unary * operators are needed in an expression, it is because an algorithm's data structures make the usage necessary. In practice, long sequences of indirections using the unary * operator are rare. Like the function call case, it may be possible to provide a visual form that provides a higher-level interpretation and hides the implementation's details of the successive indirections.

member 1031 selection

An explicit unary * operator is not the only way of specifying an indirection. Both the array subscript, `[]`, and member selection, `->`, binary operators imply an indirection. Developers rarely use the form `(*s).m` (`(&s)->m`), the form `s->m` (`s.m`) being much more obvious and natural. While the expression `s1->m1->m2->m3` is technically equivalent to `(*(*(*s1).m1).m2).m3`, it is comprehended in a left-to-right order.

Usage

SPEC 0 benchmarks

A study by Mock, Das, Chambers, and Eggers[949] looked at how many different objects the same pointer dereference referred to during program execution (10 programs from the SPEC95 and SPEC2000 benchmarks were used). They found that in 90% to 100% of cases (average 98%) the set of objects pointed at, by a particular pointer dereference, contained one item. They also performed a static analysis of the source using a variety of algorithms for deducing points-to sets. On average (geometric mean) the static points to sets were 3.3 larger than the dynamic points to sets.

If the operand points to a function, the result is a function designator; 1096

Commentary

The operand could be an object, with some pointer to function type, or it could be an identifier denoting a function that has been implicitly converted to a pointer to function type. This result is equivalent to the original function designator. Depending on the context in which it occurs this function designator may be converted to a pointer to function type.

function 731 designator

function 732 designator converted to type

C++

The C++ Standard also specifies (5.3.1p1) that this result is an lvalue. This difference is only significant for reference types, which are not supported by C.

Other Languages

Those languages that support some form of pointers to functions usually only provide a mechanism for, indirect, calls to the designated value. Operators for obtaining the function designator independent of a call are rarely provided. Some languages (e.g., Algol 88, Lisp) provide a mechanism for defining anonymous functions in an expression context, which can be assigned to objects and subsequently called.

Common Implementations

For most implementations the result is an address of a storage location. Whether there is a function definition (translated machine code) at that address is not usually relevant until an attempt is made to call the designated function (using the result).

Coding Guidelines

Because of the implicit conversions a translator is required to perform, the unary * operator is not required to cause the designated function to be called. There are a number of situations that can cause such usage to appear in source code: the token sequence may be in automatically generated source code, or the sequence may occur in developer-written source via arguments passed to macros, or developers may apply it to objects having a pointer to function type because they are unaware of the implicit conversions that need to be performed.

Example

```
1  extern void f(void);
2  extern void (*p_f)(void);
3
4  void g(void)
5  {
6  f();
7  (*f)();
8  (*******f)();
9  (*p_f)();
10 }
```

1097 if it points to an object, the result is an lvalue designating the object.

Commentary

The indirection operator produces a result that allows the pointed-to object to be treated like an anonymous object. The result can appear in the same places that an identifier (defined to be an object of the same type) can appear. The resulting lvalue might not be a modifiable lvalue. There may already be an identifier that refers to the same object. If two or more different access paths to an object exist, it is said to be *aliased*.

724 modifiable lvalue
971 object aliased

Common Implementations

Some processors (usually CISC) have instructions that treat their operand as an indirect reference. For instance, an indirect load instruction obtains its value from the storage location pointed to by the storage location that is the operand of the instruction.

1098 If the operand has type "pointer to *type*", the result has type "*type*".

unary * result type

Commentary

The indirection operator removes one level of pointer from the operand's type. The operand is required to have pointer type. In many contexts the result type of a pointer to function type will be implicitly converted back to a pointer type.

1089 unary * operand has pointer type
732 function designator converted to type

1099 If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined.[84)]

Commentary

The standard does not provide an all-encompassing definition of what an *invalid value* is. The footnote gives some examples. An invalid value has to be created before it can be assigned and this may involve a conversion operation. Those pointer conversions for which the standard defines the behavior do not create

1114 footnote 84
743 pointer to void converted to/from

invalid values. So the original creation of the invalid value, prior to assignment, must also involves undefined behavior.

object 461 initial value indeterminate

If no value has been assigned to an object, it has an indeterminate value.

array subscript 989 identical to

The equivalence between the array access operator and the indirection operator means that the behavior of what is commonly known as an *out of bounds array access* is specified here.

C++

The C++ Standard does not explicitly state the behavior for this situation.

Common Implementations

For most implementations the undefined behavior is decided by the behavior of the processor executing the program. The root cause of applying the indirection operator to an invalid valid is often a fault in a program and implementations that perform runtime checks sometimes issue a diagnostic when such an event occurs. (Some vendors have concluded that their customers would not accept the high performance penalty incurred in performing this check, and they don't include it in their implementation.) The result can often be manipulated independently of whether there is an object at that storage location, although some processors do perform a few checks.

pointer 454 cause undefined behavior

Techniques for detecting the dereferencing of invalid pointer values usually incur a significant runtime overhead[65, 681, 690, 1032, 1288] (programs often execute at least a factor of 10 times slower). A recent implementation developed by Dhurjati and Adve[351] reported performance overheads in the range 12% to 69%.

Coding Guidelines

guidelines 0 not faults

This usage corresponds to a fault in the program and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1). 1100

6.5.3.3 Unary arithmetic operators

Constraints

The operand of the unary + or - operator shall have arithmetic type; 1101

Commentary

The unary - operator is sometimes passed as a parameter in a macro invocation. In those cases where negation of an operand is not required (in the final macro replacement), the unary + operator can be passed as an argument (empty macro arguments cause problems for some preprocessors). The symmetry of having two operators can also simplify the automatic generation of source code. While it would have been possible to permit the unary + operator to have an operand of any type (since it has no effect other than performing the integer promotions on its operand), it is very unlikely that this operator would ever appear in a context that the unary - operator would not also appear in.

C++

The C++ Standard permits the operand of the unary + operator to have pointer type (5.3.1p6).

Coding Guidelines

While applying the unary minus operator to an operand having an unsigned integer type is seen in some algorithms (it can be a more efficient method of subtracting the value from the corresponding `U*_MAX` macro, in **`<limits.h>`**, and adding one), this usage is generally an oversight by a developer.

Rev 1101.1 The promoted operand of the unary - operator shall not be an unsigned type.

1102 of the ~ operator, integer type;

Commentary

There are algorithms (e.g., in graphics applications) that require the bits in an integer value to be complemented, and processors invariably contain an instruction for performing this operation. Complementing the bits in a floating-point value is a very rarely required operation and processors do not contain such an instruction. This constraint reflects this common usage.

Other Languages

While many languages do not contain an equivalent of the ~ operator, their implementations sometimes include it as an extension.

Coding Guidelines

Some coding guideline documents only recommend against the use of operands having a signed type. The argument is that the representation of unsigned types is defined by the standard, while signed types might have one of several representations. In practice, signed types almost universally have the same representation—two's complement. However, the possibility of variability of integer representation across processors is not the only important issue here. The ~ operator treats its operand as a sequence of bits, not a numeric value. As such it may be making use of representation information and the guideline recommendation dealing with this issue would be applicable.

612 two's complement
569.1 representation information using

1103 of the ! operator, scalar type.

! operand type

Commentary

The logical negation operator is defined in terms of the equality operator, whose behavior in turn is only defined for scalar types.

1113 ! equivalent to
1213 equality operators constraints

C++

The C++ Standard does not specify any requirements on the type of the operand of the ! operator.

> *The operand of the logical negation operator* **!** *is implicitly converted to* **bool** *(clause 4);* 5.3.1p8

But the behavior is only defined if operands of scalar type are converted to **bool**:

> *An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type* **bool**. 4.12p1

Other Languages

Some languages require the operand to have a boolean type.

Coding Guidelines

The following are two possible ways of thinking about this operator are:

1. As a shorthand form of the != operator in a conditional expression. That is, in the same way the two forms `if (x)` and `if (x == 0)` are equivalent, the two forms `if (!x)` and `if (x != 0)` are equivalent.
2. As a logical negation operator that reverses the state of a boolean value (it can take as its operand a value in either of the possible boolean representation models and map it to the model that uses the 0/1 for its boolean representation).

476 boolean role

A double negative is very often interpreted as a positive statement in English (e.g., "It is not unknown for double negatives to occur in C source"). The same semantics that apply in C. However, in some languages

(e.g., Spanish) a double negative is interpreted as making the statement more negative (this usage does occur in casual English speech, e.g., "you haven't seen nothing yet", but it is rare and frowned on socially[121]).

The token `!` is commonly called the *not* operator. This term is a common English word whose use in a sentence is similar to its use in a C expression. Through English language usage the word *not*, or an equivalent form, can appear as part of an identifier spelling (e.g., `not_finished`, `no_signal`, or `unfinished`). The use of such identifiers in an expression can create a double negative (e.g., `!not_finished` or `not_finished != 1`).

English 792 negation

A simple expression containing a double negation is likely to require significantly more cognitive resources to comprehend than a one that does not. Changing the semantic associations of an identifier from (those implied by) `not_finished` to `finished` would require that occurrences of `not_finished` be changed to `!finished` (plus associated changes to any appearances of the identifier as the operand of the `!` or the equality operators).

Calculating the difference in cognitive cost/benefit between using an identifier spelling that represents a negated form and one that does not requires information on a number of factors. For instance, whether any double negative forms actually appear in the source, the extent to which the *not* spelling form provides a good fit to the application domain, and any cognitive cost differences between the alternative forms `not_finished` and `!finished`. Given the uncertainty in the cost/benefit analysis no guideline recommendation is given here.

Table 1103.1: Occurrence of the unary `!` operator in various contexts (as a percentage of all occurrences of this operator and the percentage of all occurrences of the given context that contains this operator). Based on the visible form of the `.c` files.

Context	% of `!`	% of Contexts
`if` control-expression	91.0	17.4
`while` control-expression	2.3	8.2
`for` control-expression	0.3	0.7
`switch` control-expression	0.0	0.0
other contexts	6.4	—

Semantics

The result of the unary + operator is the value of its (promoted) operand. 1104

Commentary

Also, the result of the unary **+** is not an lvalue.

Other Languages

Many languages do not include a unary **+** operator.

Common Implementations

Early versions of K&R C treated `p=+2` as being equivalent to `p+=2`.[723]

Coding Guidelines

One use of the unary **+** operator is to remove the lvalue-ness of an object appearing as an argument in a macro invocation. This usage guarantees that the object passed as an argument cannot be modified or have its address taken.

Use of the unary **+** operator is very rare in developer-written source. If it appears immediately after the **=** operator in existing code, the possible early K&R interpretation might be applicable. The usage is now sufficiently rare that a discussion on whether to do nothing, replace every occurrence by the sequence **+=**, introduce a separating white-space character, parenthesize the value being assigned, or do something else is not considered worthwhile.

Example

```
/*
 * Sum up to three values.  Depending on the arguments
 * passed + is either a unary or binary operator.
 */
#define ADD3(a, b, c)    (a + b + c + 0)

        ADD3(1, 2, 3) => (1 + 2 + 3 + 0)
        ADD3(1, 2,  ) => (1 + 2 +   + 0)
        ADD3(1,  , 3) => (1 +   + 3 + 0)
        ADD3(1,  ,  ) => (1 +   +   + 0)
        ADD3( ,  ,  ) => (  +   +   + 0)
```

1105 The integer promotions are performed on the operand, and the result has the promoted type.

Commentary

The two contexts in which the integer promotions would not be performed, unless the unary + operator is applied, are the right operand of a simple assignment and the operand of the **`sizeof`** operator.

1303 simple assignment
1119 sizeof result of

1106 The result of the unary - operator is the negative of its (promoted) operand.

Commentary

The expression `-x` is not always equivalent to `0-x`; for instance, if `x` has the value `0.0`, the results will be `-0.0` and `0.0`, respectively.

Common Implementations

Most processors include a single instruction that performs the negation operation. On many RISC processors this instruction is implemented by the assembler using an alias of the subtract instruction (for integer operands only). On such processors there is usually a register hardwired to contain the value zero (the IBM/Motorola PowerPC[969] does not); subtracting the operand from this register has the required effect. For IEC 60559 floating-point representations, the negation operator simply changes the value of the sign bit.

Coding Guidelines

If the operand has an unsigned type, the result will always be a positive or zero value. This issue is discussed elsewhere.

1101.1 unary minus unsigned operand

Example

The expression `-1` is the unary `-` operator applied to the integer constant `1`.

1107 The integer promotions are performed on the operand, and the result has the promoted type.

Commentary

Because unary operators have a single operand, it is not necessary to perform the usual arithmetic conversions (the integer promotions are performed on the operands of the unary operators for the same reason).

706 usual arithmetic conversions
675 integer promotions

Coding Guidelines

The integer promotions may convert an unsigned type to a signed type. However, this can only happen if the signed type can represent all of the values of the unsigned type. This is reflected in the guideline recommendation for unsigned types.

715 signed integer represent all unsigned integer values
1101.1 unary minus unsigned operand

1108 The result of the ~ operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set).

bitwise complement result is

Commentary

bitwise not, logical negation 1111 result is

The term *bitwise not* is sometimes used to denote this operator (it is sometimes also referred to by the character used to represent it, tilde). Because its use is much less frequent than logical negation, this term is rarely shortened.

Common Implementations

Most processors have an instruction that performs this operation. An alternative implementation is to exclusive-or the operand with an all-bits-one value (containing the same number of bits as the promoted type). The Unisys A Series[1390] uses signed magnitude representation. If the operand has an unsigned type, the sign bit in the object representation (which is treated as a padding bit) is not affected by the bitwise complement operator. If the operand has a signed type, the sign bit does take part in the bitwise complement operation.

Example

```
1  V &= ~BITS; /* Clear the bits in V that are set in BITS. */
```

The integer promotions are performed on the operand, and the result has the promoted type. 1109

Commentary

Performing the integer promotions can increase the number of representation bits used in the value that the complement operator has to operate on.

Coding Guidelines

The impact of the integer promotions on the value of the result is sometimes overlooked by developers. During initial development these oversights are usually quickly corrected (the results differ substantially from the expected range of values and often have a significant impact on program output). Porting existing code to a processor whose **`int`** size differs from the original processor on which the code executed can cause latent differences in behavior to appear. For instance, if `sizeof(int)==sizeof(short)` on the original processor, then any integer promotions on operands having type **`short`** would not increase the number of bits in the value representation and a program may have an implicit dependency on this behavior occurring. Moving to a processor where `sizeof(int) > sizeof(short)` may require modifications to explicitly enforce this dependency. The issues involved in guideline recommendations that only deliver a benefit when a program is ported to a processor whose integer widths are different from the original processor are discussed elsewhere.

coding guidelines 0 the benefit

Example

```
1  unsigned char uc = 2;
2  signed char sc = -1;
3
4  void f(void)
5  {
6  /*
7   * Using two's complement notation the value 2 is represented
8   * in the uc object as the bit pattern 00000010
9   * If int is represented using 16 bits, this value is promoted
10  * to the representation     0000000000000010
11  * after being complemented  1111111111111101 is the result
12  */
13            ~uc             ;
14 /*
15  * Using two's complement notation the value -1 is represented
16  * in the sc object as the bit pattern 11111110
17  * If int is represented using 16 bits, this value is promoted
```

```
18    * to the representation     1111111111111110
19    * after being complemented  0000000000000010 is the result
20    */
21              ~sc             ;
22    }
```

1110 If the promoted type is an unsigned type, the expression **~E** is equivalent to the maximum value representable in that type minus **E**.

Commentary

This is the standard pointing out an equivalence that holds for the binary representation of integer values.

C++

The C++ Standard does not point out this equivalence.

Coding Guidelines

The issues surrounding the use of bitwise operations to perform equivalent arithmetic operations is discussed elsewhere.

945 bitwise operators

1111 The result of the logical negation operator **!** is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0.

logical negation result is

Commentary

The term *not* (or *logical not*) is often used to denote this operator. The much less frequently used operator, bitwise complement, takes the longer name.

logical not
1108 bitwise complement result is

C++

its value is **true** *if the converted operand is* **false** *and* **false** *otherwise.* 5.3.1p8

This difference is only visible to the developer in one case. In all other situations the behavior is the same false and true will be converted to 0 and 1 as-needed.

1112 logical negation result type

Other Languages

Languages that support a boolean data type usually specify **true** and **false** return values for these operators.

Common Implementations

The implementation of this operator often depends on the context in which it occurs. The machine code generated can be very different if the result value is used to decide the control flow (e.g., it is the final operation in the evaluation of a controlling expression) than if the result value is the operand of further operators. In the control flow case an actual value of 0 or 1 is not usually required. On many processors loading a value from storage into a register will set various bits in a conditional flags register (these flag bit settings usually specify some relationship between the value loaded and zero— e.g., equal to, less than, etc.). A processor's conditional branch instructions use the current settings of combinations of these bits to decide whether to take the branch or not. When the result is used as an operand in further operations, a 0 or 1 value is needed; the generated machine code is often more complex. A common solution is the following pseudo machine code sequence (which leaves the result in REG_1):

1744 if statement operand compare against 0

```
1   load REG_1, 0
2   load REG_2, Operand
3   Branch_over_next_instr_if_last_load_not_zero
4   load REG_1, 1
```

The instruction Branch_over_next_instr_if_last_load_not_zero may be a single instruction, or several instructions.

Coding Guidelines

While the result is specified in numeric terms, most occurrences of this operator are as the top-level operator in a controlling expression (see Usage below). These contexts are usually considered in boolean rather than numeric terms.

boolean role 476

---

logical negation result type

The result has type **int**. 1112

Commentary

The C90 Standard did not include a boolean data type and C99 maintains compatibility with this existing definition.

C++

5.3.1p8 *The type of the result is* ***bool****.*

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

Other Languages

In languages that support a boolean type the result of logical operators usually has a boolean type.

Coding Guidelines

The possible treatment of the result of logical, and other operators as having a boolean type, is discussed elsewhere.

boolean role 476

! equivalent to

The expression **!E** is equivalent to **(0==E)**. 1113

Commentary

null pointer 748 constant

In the case of pointers the 0 is equivalent to the null pointer constant. In the case of E having a floating-point type, the constant 0 will be converted to 0.0. The standard is being idiosyncratic in expressing this equivalence (the form (E==0) is more frequent, by a factor of 28, in existing code).

C++

There is no explicit statement of equivalence given in the C++ Standard.

Common Implementations

logical 1111 negation result is

Both forms occur sufficiently frequently, in existing code, that translator implementors are likely to check for the context in which they occur in order to generate the appropriate machine code.

Coding Guidelines

visually compact code 0 efficiency belief punctuator 912 syntax

Both forms are semantically identical, and it is very likely that identical machine code will be generated for both of them. (The sometimes-heard rationale of visual complexity of an expression being equated to inefficiency of program execution is discussed elsewhere.) Because of existing usage (the percentage occurrence of both operators in the visible source is approximately comparable) developers are going to have to learn to efficiently comprehend expressions containing both operators. Given this equivalence and existing practice, is there any benefit to be gained by a guideline recommending one form over the other? Both have their disadvantages:

- The **!** character is not frequently encountered in formal education, and it may be easy to miss in a visual scan of source (no empirical studies using the **!** character are known to your author).

controlling 1740 expression if statement

- The equality operator, **==**, is sometimes mistyped as an assignment operator, **=**.

Perhaps the most significant distinguishing feature of these operators is the conceptual usage associated with their operand. If this operand is primarily thought about in boolean terms, the conceptually closest operator is `!`. If the operand is thought of as being arithmetic, the conceptually closest operator is `==`. 476 boolean role

A number of studies have investigated the impact of negation in reasoning tasks. In natural languages negation comes in a variety of linguistic forms (e.g., "no boys go to class", "few boys go to class", "some boys go to class") and while the results of these studies[696] of human performance using these forms may be of interest to some researchers, they don't have an obvious mapping to C language usage (apart from the obvious one that negating a sentence involves an additional operator, the negation, which itself needs cognitive resources to process).

Usage

The visible form of the `.c` files contain 95,024 instances of the operator `!` (see Table 912.2 for information on punctuation frequencies) and 27,008 instances of the token sequence `== 0` (plus 309 instances of the form `== 0x0`). Integer constants appearing as the operand of a binary operator occur 28 times more often as the right operand than as the left operand.

1114 84) Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. footnote 84

Commentary

This footnote sentence should really have been referenced from a different paragraph, where these equivalences are discussed. 1092 &*

C90

This equivalence was not supported in C90, as discussed in the response to DR #012, #076, and #106.

C++

At the moment the C++ Standard specifies no such equivalence, explicitly or implicitly. However, this situation may be changed by the response to DR #232.

1115 It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `*&E` is a function designator or an lvalue equal to `E`. *&

Commentary

This statement can be deduced from the specifications of the two operators concerned.

1116 If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Commentary

The result of the cast operator is not an lvalue. However, if the operand is a pointer, the pointed-to object does not lose its lvalue-ness. This sentence simply points out the type of the result of the operations and its lvalue-ness; it does not give any additional semantics to the cast or dereference. 1131 footnote 85

C++

The C++ Standard makes no such observation.

1117 Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Commentary

This list contains some examples of invalid values that may appear directly in the source; it is not exhaustive (another example is dereferencing a pointer-to function). The invalid values may also be the result of an operation that has undefined behavior. For instance, using pointer arithmetic to create an address that does

not correspond to any physical memory location supported by a particular computing system. (In virtual memory systems this case would correspond to an unmapped address.)

C90

The wording in the C90 Standard only dealt with the address of objects having automatic storage duration.

C++

The C++ Standard does not call out a list of possible invalid values that might be dereferenced.

Other Languages

Most other languages do not get involved in specifying such low-level details, although their implementations invariably regard the above values as being invalid.

Common Implementations

A host's response to an attempt to use an invalid pointer value will usually depend on the characteristics of the processor executing the program image. In some cases an exception which can be caught by the program, may be signaled. The extent to which a signal handler can recover from the exception will depend on the application logic and the type of invalid valid dereference.

On many implementations the `offsetof` macro expands to an expression that dereferences the null pointer.

Coding Guidelines

coding guidelines background to 0

One of the ways these guideline recommendations attempt to achieve their aim is to attempt to prevent invalid values from being created in the first place.

6.5.3.4 The `sizeof` operator

Constraints

---

sizeof constraints

The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. 1118

Commentary

In the C90 Standard the result of the **`sizeof`** operator was a constant known at translation time. While there are some applications where being able to find out the size of a function would be useful (one specification might be the number of bytes in its generated machine code), this information is not of general utility. The C90 constraint was kept.

If the **`sizeof`** operator accepted a bit-field as an operand, it would have to return a value measured in bits for all its operands.

sizeof result of 1119

C++

The C++ Standard contains a requirement that does not exist in C.

5.3.3p5 *Types shall not be defined in a **`sizeof`** expression.*

A C source file that defines a type within a **`sizeof`** expression is likely to cause a C++ translator to issue a diagnostic. Defining a type within a **`sizeof`** expression is rarely seen in C source.

```
1  int glob = sizeof(enum {E1, E2}); /* does not affect the conformance status of the program */
2                                    // ill-formed
```

Other Languages

A few other languages provide an explicit mechanism (e.g., an operator or a keyword) for obtaining the number of bytes occupied by an object. Such a mechanism is also a common extension in implementations of languages that do not specify one. Other languages obtain the size implicitly in those contexts where it is needed (i.e., the operand in a call to **new** memory allocation function in Pascal).

Table 1118.1: Occurrence of the **sizeof** operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
struct	48.2	**unsigned short**	2.7
[ ]	12.2	**struct ***	2.6
int	11.6	**char**	2.0
other-types	4.7	**unsigned char**	1.5
long	3.8	**char ***	1.5
unsigned int	3.6	**signed int**	1.2
unsigned long	3.4	**union**	1.1

Semantics

1119 The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type.

sizeof result of

Commentary

The operand referred to is the execution-time value of the operand. In the case of string literals, escape sequences will have been converted to a single or multibyte character. In these cases the value returned by the **sizeof** operator does not correspond to the number of characters visible in the source code. Most of the uses of the result of this operator work at the byte, not the bit, level; for instance, the argument of a memory-allocation function, which operates in units of bytes. Having to divide the result by CHAR_BIT, for most uses, would not be worth the benefit of being able to accept bit-field members.

133 translation phase 5

Other Languages

The **SIZE** attribute in Ada returns the number of bits allocated to hold the object, or type. The **BIT_SIZE** intrinsic in Fortran 90 returns the number of bits in its integer argument; the **SIZE** intrinsic returns the number of elements in an array.

Common Implementations

A few vendors have extended the **sizeof** operator. For instance, Diab Data[353] supports a second argument to the parenthesized form of the **sizeof** operator. The value of this argument changes the information returned (e.g., if the value of the second argument is 1 the alignment of the type is returned, if it is 2 a unique value denoting the actual type is returned).

Coding Guidelines

The size of an object, or type, is representation information and the guideline recommendation dealing with the use of representation information might be thought to be applicable. However, in some contexts many uses of the **sizeof** operator are symbolic. The contexts in which the size of an operand is often used include the following:

569.1 representation information

822 symbolic name using

- A call to a storage allocation function requires the number of bytes to allocate.
- When copying the representation of an object, either to another object or to a binary file, the number of bytes to be copied is required.
- When an object is being overlaid over the same storage as another object (using a union or pointer to object type), the sizes in the two types need to agree.

- When calculating the range of values representable by the operand (based on the number of bits it contains).

In some of the uses in these contexts the result of the **sizeof** operator is treated as a symbolic value— the size of its operand, with no interest in its numeric properties. While in others the result is manipulated as an arithmetic value; it is an intermediate value used in the calculation of the final value. However, a strong case can be made for claiming that certain kinds of arithmetic operations are essentially symbolic in nature:

- Multiplication of the result (e.g, to calculate the size of an array of objects)
- Division of the result (e.g., to calculate how many objects will fit in a given amount of storage)
- Subtracting from the result (e.g., to calculate the offset of the character that is third from the end of a string literal.
- Adding to the result (e.g., calculating the size of an array needed to hold several strings)

Dev 569.1 The **sizeof** operator may be used provided the only operators applied to its result (and the result of these operations) are divide and multiple.

Dev 569.1 The **sizeof** operator whose operand has an array type may be used provided the only operators applied to its result (and the result of these operations) are divide, multiply, addition, and subtraction.

For simplicity the deviation wording permits some unintended uses of representation information. For instance, the deviations permit both of the expressions `sizeof(array_of_int)-5` and `sizeof(array_of_char)-5`. There is a difference between the two in that in the former case the developer is either making use of representation information or forgot to write `sizeof(array_of_int)-5*sizeof(int)` (these guideline recommendations are not intended to recommend against constructs that are faults). Character types are special in that `sizeof(char)` is required to be 1, so it is accepted practice to omit the multiplication for these types.

guidelines 0 not faults
sizeof char 1124 defined to be 1

The size is determined from the type of the operand. 1120

Commentary

If an object appears as the operand, its declared type is used, not its effective type. The operand is also a special case in that some implicit conversions do not occur in this context.

For floating-types, any extra precision maintained by an implementation is not included in the number of bytes returned. For instance, `sizeof(a_double * b_double)` always returns the size of the type specified by the C semantics, not the size of the representation used by the implementation when multiplying two objects of type **double**.

lvalue 725 converted to value
array 729 converted to pointer
function 732 designator converted to type
FLT_EVAL_METHOD 354

C++

5.3.3p1 *The **sizeof** operator yields the number of bytes in the object representation of its operand.*

Coding Guidelines

Developers sometimes write code that uses an operand's size to deduce the range of values it can represent (applies to integer types only). Information on the range of values representable in a type is provided by the contents of the header **<limits.h>**. However, when writing a macro having an argument that can be one of several types, there is no mechanism for deducing the type actually passed. It is not possible to use any of

the macros provided by the header **<limits.h>**. The **sizeof** operator provides a solution that works on the majority of processors.

A size determined from the type of the operand need not provide an accurate indication of the range of values representable in that operand type (it provides an upper bound on the range of values that can be stored in an object of that type). A type may contain padding bytes, which will be included in its size. In the case of floating-point types, it is also possible that an expression is evaluated to a greater precision than implied by its type. Using the **sizeof** operator for this purpose is covered by the guideline recommendation dealing with the use of representation information.

354 FLT_EVAL_METH
569.1 representation information using

1121 The result is an integer.

Commentary

To be exact, the result has an integer type, `size_t`.

C90

In C90 the result was always an integer constant. The C99 contexts in which the result is not an integer constant all involve constructs that are new in C99.

C++

Like C90, the C++ Standard specifies that the result is a constant. The cases where the result is not a constant require the use of types that are not supported by C++.

1122 If the type of the operand is a variable length array type, the operand is evaluated;

sizeof operand evaluated

Commentary

The number of elements in the variable length array is not known until its index expression is evaluated. This evaluation may cause side effects. The requirement specified in this C sentence is weakened by a later sentence in the standard. It is possible that the operand may only be partially evaluated.

1584 sizeof VLA unspecified evaluation

C90

Support for variable length array types is new in C99.

C++

Variable length array types are new in C99 and are not available in C++.

Coding Guidelines

The issue of side effects in VLA's is discussed elsewhere.

1584 sizeof VLA unspecified evaluation

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  int loc = sizeof(int [glob++]); /* glob is incremented. */
6
7  /*
8   * To obtain the size of the type ++glob need not be evaluated in
9   * the first case, but must be evaluated in the second.
10  */
11 loc=sizeof((int *)[++glob]);
12 loc=sizeof( int * [++glob]);
13 }
```

1123 otherwise, the operand is not evaluated and the result is an integer constant.

sizeof operand not evaluated

Commentary

A full expression having a **sizeof** operator as its top-level operator, with such an operand, can occur anywhere that an integer constant can occur. The size is obtained from the type of the operand. This information is available during translation. (There is no need to generate any machine code to evaluate the operand, and this requirement prohibits such generation.) Although the operand is not evaluated, any operators that appear in it will still cause the integer promotions and usual arithmetic conversions to be performed.

integer promotions 675
usual arithmetic conversions 706

Coding Guidelines

Some coding guideline documents recommend that the operand of the **sizeof** operator should not contain any side effects. In practice such usage is very rarely seen and no such guideline recommendation is given here.

Example

```
1  extern int i;
2  extern unsigned char uc;
3
4  void f(void)
5  {
6  int loc_i = sizeof(i++);    /* i is not incremented. */
7  int loc_uc = sizeof(uc);    /* sizeof an unsigned char. */
8  int loc_i_uc = sizeof(+uc); /* Operand promoted first. */
9  }
```

---

sizeof char defined to be 1

When applied to an operand that has type **char**, **unsigned char**, or **signed char**, (or a qualified version thereof) the result is 1. 1124

Commentary

The number of bits in the representation of a character type is irrelevant. By definition the number of bytes in a character type is one.

byte 53
addressable unit

Coding Guidelines

Developers sometimes associate a byte as always containing eight bits. On hosts where the character type is 16 bits, this can lead to the incorrect assumption that applying **sizeof** to a character type will return the value 2. These issues are discussed elsewhere.

CHAR_BIT 307
macro

When applied to an operand that has array type, the result is the total number of bytes in the array.[85)] 1125

Commentary

In this case an array is not converted to a pointer to its first element.

array 729
converted to pointer

When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. 1126

Commentary

All of the bytes in the object representation need to be included because of the kinds of use to which the result of the **sizeof** operator is put (e.g., allocating storage and copying objects). Trailing padding needs to be taken into account because more than one object of that type may be allocated or copied (e.g., an array of types having a given size, or a structure containing a member having a flexible array type). The standard requires that there be no leading padding.

structure 1428
trailing padding
structure 1424
unnamed padding

1127 The value of the result is implementation-defined, and its type (an unsigned integer type) is **size_t**, defined in **<stddef.h>** (and other headers). sizeof result type

Commentary

The implementation-defined value will be less than or equal to the value of the SIZE_MAX macro. There is no requirement that SIZE_MAX == PTRDIFF_MAX. When the operand of **sizeof** contains more bytes than can be represented in the type size_t (e.g., char x[SIZE_MAX/2][SIZE_MAX/2];). The response to DR #266 stated:

The committee has deliberated and decided that more than one interpretation is reasonable. DR #266

There is no requirement on implementations to provide a definition of the type size_t that is capable of representing the number of bytes in any object that the implementation is capable of allocating storage for. It is the implementation's responsibility to ensure that the type it uses for size_t internally is the same as the typedef definition of size_t in the supplied header, **<stddef.h>**. If these types differ, the implementation is not conforming.

A developer can define a typedef whose name is size_t (subject to the constraints covering declarations of identifiers). Such a declaration does not affect the type used by a translator as its result type for the **sizeof** operator.

C++

. . . ; the result of ***sizeof*** *applied to any other fundamental type (3.9.1) is implementation-defined.* 5.3.3p1

The C++ Standard does not explicitly specify any behavior when the operand of **sizeof** has a derived type. A C++ implementation need not document how the result of the **sizeof** operator applied to a derived type is calculated.

Coding Guidelines

Use of the **sizeof** operator can sometimes produce results that surprise developers. The root cause of the surprising behavior is usually that the developer forgot that the result of the **sizeof** has an unsigned type (which causes the type of the other operand, of a binary operator, to be converted to an unsigned type). Developers forgetting about the unsignedness of the result of a **sizeof** is not something that can be addressed by a guideline recommendation.

1128 EXAMPLE 1 A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the **alloc** function should ensure that its return value is aligned suitably for conversion to a pointer to **double**.

Commentary

Measurements of existing source (see Table 1080.1) shows that this usage represents at most 14% of all uses of the **sizeof** operator.

1129 EXAMPLE 2 Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

Commentary

The declaration of an object having an array type may not contain an explicit value for the size, but obtain it from the number of elements in an associated initializer.

Other Languages

Some languages provide built-in support for obtaining the bounds or the number of elements in an array. For instance, Fortran has the intrinsic functions **LBOUND** and **UBOUND**; Ada specifies the attributes **first** and **last** to return the lower and upper bounds of array, respectively.

EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function: 1130

```
#include <stddef.h>

size_t fsize3(int n)
{
        char b[n+3];       // variable length array
        return sizeof b;   // execution time sizeof
}

int main()
{
        size_t size;
        size = fsize3(10); // fsize3 returns 13
        return 0;
}
```

Commentary

In this example the result of the **sizeof** operator is not known at translation time.

C90

This example, and support for variable length arrays, is new in C99.

---

footnote 85

85) When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1). 1131

Commentary

array type 1598 adjust to pointer to array 729 converted to pointer

The more specific reference is 6.7.5.3. The parameter type is converted before the **sizeof** operator operates on it. There is no array type for the operand exception.

C++

This observation is not made in the C++ Standard.

Other Languages

Converting an array parameter to a pointer type is unique to C (and C++).

Coding Guidelines

qualified 1571 array of

Traditionally, the reason for declaring a parameter to have an array type is to create an association in the developer's mind and provide hints to static analysis tools. (Functionality added in C99 provides a mechanism for specifying additional semantics with this usage.) In most contexts it does not matter whether readers of the code treat the parameter as having an array or pointer type. However, in the context of an operand to the **sizeof** operator, there is an important difference in behavior.

Example

In the following the array `b` will be declared to have an upper bound of `sizeof(int *)`, not the number of bytes in the array `a`.

```
1  void f(int a[3])
2  {
```

```
3  unsigned char b[sizeof(a)];
4  }
```

1132 **Forward references:** common definitions <stddef.h> (7.17), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.6), array declarators (6.7.5.2).

6.5.4 Cast operators

cast-expression syntax

1133

cast-expression:
unary-expression
(*type-name* **)** *cast-expression*

Commentary

A *cast-expression* is also a unary operator. Given that the evaluation of a sequence of unary operators always occurs in a right-to-left order, the lower precedence of the cast operator is not significant.

C++

The C++ Standard uses the terminal name *type-id*, not *type-name*.

Other Languages

Some languages parenthesize the *cast-expression* and leave the *type-name* unparenthesized.

Usage

Measurements by Stiff, Chandra, Ball, Kunchithapadam, and Reps[1301] of 1.36 MLOC (SPEC95 version of gcc, binutils, production code from a Lucent Technologies product and a few other programs) showed a total of 23,947 casts involving 2,020 unique types. Of these 15,704 involved scalar types (not involving a structure, union, or function pointer) and 447 function pointer types. Of the remaining casts 7,796 (1,276 unique types) involved conversions between pointers to **void/char** and pointers to structure (in either direction) and 1,053 (209 unique types) conversions between pointers to structs.

Constraints

1134 Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

cast scalar or void type

Commentary

Casting to the **void** type is a method of explicitly showing that the value of the operand is discarded. Casting a value having a structure or union type has no obvious meaning. (Would corresponding member names be assigned to each other? What would happen to those members that did not correspond to a member in the other type?)

C++

There is no such restriction in C++ (which permits the type name to be a class type). However, the C++ Standard contains a requirement that does not exist in C.

Types shall not be defined in casts. 5.4p3

A C source file that defines a type within a cast is likely to cause a C++ translator to issue a diagnostic (this usage is rare).

```
1  extern int glob;
2
```

```
3  void f(void)
4  {
5  switch ((enum {E1, E2, E3})glob) /* does not affect the conformance status of the program */
6                                   // ill-formed
7     {
8     case E1: glob+=3;
9              break;
10    /* ... */
11    }
12 }
```

Other Languages

Some languages require the cast to have an arithmetic type. Algol 68 permits a cast to any type for which an assignment would be permitted, and nothing else (e.g., if `T var; var := value` is permitted, then `value` can be cast to type T).

Common Implementations

gcc supports the casting of scalar types to union types. The scalar type must have the same type as one of the members of the union type. The cast is treated as being equivalent to assigning to the member having that type. This extension removes the need to know the name of the union member.

```
1  union T {
2           int mem_1;
3           double mem_2;
4          } u;
5  int x;
6  double y;
7
8  void f(void)
9  {
10 u = (union T)x;       /* gcc: equivalent to u.mem_1 = x */
11 u = (__typeof__(u))y; /* gcc: equivalent to u.mem_2 = y */
12 }
```

Coding Guidelines

In this discussion a suffixed literal will be treated as an explicit cast of a literal value, while an unsuffixed literal is not treated as such. An explicit cast is usually interpreted as showing that the developer intended the conversion to take place. It is taken as a statement of intent. It is often assumed, by readers of the source, that an explicit cast specifies the final type of the operand. An explicit cast followed by an implicit one is suspicious; it suggests that either the original developer did not fully understand what was occurring or that subsequent changes have modified the intended behavior.

Cg 1134.1 The result of a cast operation shall not be implicitly converted to another type.

Dev 1134.1 If the result of a macro substitution has the form of an expression, that expression may be implicitly converted to another type.

Example

```
1  #include "stuff.h"
2
3  #define MAX_THINGS 333333u
4  #define T_VAL ((int)x)
```

```
5
6   extern SOME_INTEGER_TYPE x;
7
8   void f(void)
9   {
10  long num_1_things = MAX_THINGS;
11  long num_2_things = (long)MAX_THINGS;
12  short num_3_things = (short)MAX_THINGS;
13
14  long count_1_things = (long)44444u;
15  short count_2_things = (short)44444u;
16
17  long things_1_val = x;
18  long things_2_val = (long)x;
19  long things_3_val = (long)(int)x;
20  long things_4_val = (long)T_VAL;
21  }
```

Usage

Usage information on implicit conversions is given elsewhere (see Table 653.1).

Table 1134.1: Occurrence of the cast operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
(other-types)	other-types	40.1	(char *)	const char *	1.6
(void *)	_ int	18.9	(union *)	void *	1.5
(struct *)	struct *	11.2	(void)	long	1.3
(struct *)	_ int	4.2	(unsigned long)	unsigned long	1.3
(char *)	char *	4.0	(int)	int	1.3
(char *)	struct *	3.9	(unsigned int)	int	1.2
(struct *)	void *	2.8	(enum)	int:8 24	1.2
(unsigned char)	int	1.7	(char)	_ int	1.2
(struct *)	char *	1.7	(unsigned long)	ptr-to *	1.0

1135 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

pointer conversion constraints

Commentary

The special cases listed in 6.5.16.1 allow some values that are assigned to have their types implicitly converted to that of the object being assigned to. There are also contexts (e.g., the conditional operator) where implicit conversions occur.

1296 simple assignment constraints
1266 conditional operator second and third operands

C90

This wording appeared in the Semantics clause in the C90 Standard; it was moved to constraints in C99. This is not a difference if it is redundant.

C++

The C++ Standard words its specification in terms of assignment:

> *An expression* `e` *can be implicitly converted to a type* `T` *if and only if the declaration "*`T t=e;`*" is well-formed, for some invented temporary variable* `t` *(8.5).*

4p3

Semantics

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. 1136

Commentary

simple assignment constraints 1296
default argument promotions 1009
pointer conversion constraints 1135
_Bool converted to 680

Most of the implicit conversions that occur in C cause types to be widened. In the case of assignment and passing arguments narrowing may occur. Explicit casts are required to support conversions involving pointers. The use of the phrase *named type* is misleading in that the type converted to may be anonymous. For most pairs of types, but not all, the cast operator is commutative (assuming the operand is within the defined range of the types).

Although C has a relatively large number of integer types (compared to the single one supported in most languages), the language has been designed so that explicit casts are rarely required for these constructs. For instance, an explicit cast is required to convert an array index having a floating type or to convert an argument having type **long** to **int** when using old style function declarations. Typedef names are synonyms and do not require the use of explicit casts seen in more strongly typed languages.

typedef 1633 is synonym

Common Implementations

The most common implementation of pointer and widening integer conversions is to treat the existing value bits as having the new type (in many cases values having integer type will already be represented in a processor's register using the full number of bits).

In the case of conversions to narrower integer types, the generated machine code may depend on what operation performed on the result of the cast. In the case of assignment the appropriate least-significant bits of the unconverted operand are usually stored. For other operators implementations often zero out the nonsignificant bits (performing a bitwise-AND operation is often faster than a remainder operation) and for signed types sign extend the result.

The representation difference between integer and floating-point types is so great that many processors contain instructions that perform the conversion. In other cases internal calls to library functions have to be generated by the translator.

Coding Guidelines

Casts are sometimes used as a means of reducing a value so that it falls within a particular range of values (i.e., effectively performing a modulo operation). This usage may be driven by the desire not to use two other possible operators: (1) bitwise operators, because of the self-evident use of representation information; and (2) the remainder operator, because it is viewed as having poor performance. A cast operator used for this purpose is making use of representation information; the range of values that happen to be supported by the cast type on the implementation used. This usage is a violation of the guideline recommendation dealing with the use of representation information.

% operator 1149 result

representation information using object 569.1 480.1 int type only

Following the guideline recommendation specifying the use of a single integer type removes the need to consider many of the conversion issues that can otherwise arise with these types.

In some contexts an explicit conversion is required, while in other contexts a translator will perform an implicit conversion. Are there any benefits to using an explicit cast operator in these other contexts? An explicit cast can be thought about in several ways:

operand 653 convert automatically

- An indicator of awareness on the part of the codes author. The assumption is often made by subsequent readers of the source that an explicit cast shows that a conversion was expected/intended. Static analysis tools tend to treat implicit conversions with some suspicion.
- A change of interpretation of a numeric value (e.g., signed/unsigned or integer/floating-point conversions).
- A method of removing excess accuracy in floating-point operations.
- A method of changing the conversions that would have occurred as a result of the usual arithmetic conversions.

FLT_EVAL_METHOD 354

usual arithmetic conversions 706

- A method of explicitly indicating that a change in role, or symbolic name status, of an object is intended.

1352 object role
822 symbolic name

The cost/benefit issues and possible guideline recommendations are discussed elsewhere.

653 operand convert automatically

1137 This construction is called a *cast*.[86)]

Commentary

This defines the term *cast*. Developers often call this construction an *explicit cast*, while an implicit conversion performed by a translator is often called an *implicit cast*.

C++

The C++ Standard uses the phrase *cast notation*. There are other ways of expressing a type conversion in C++ (functional notation, or a type conversion operator). The word *cast* could be said to apply in common usage to any of these forms (when technically it refers to none of them).

Other Languages

The terms *cast* or *coercion* are often used in programming language definitions.

Coding Guidelines

The term *cast* is often used by developers to refer to the implicit conversions performed by an implementation. This is incorrect use of terminology, but there is very little to be gained in attempting to change existing, common usage developer terminology.

1138 A cast that specifies no conversion has no effect on the type or value of an expression.[87)]

Commentary

The standard does not define what is meant by *no conversion*; only one kind of cast can have no effect on the semantic type (i.e., a cast that has the same type as the expression it operates on). If an implementation uses the same representation for two types, any cast between those two types will have no effect on the value. A cast to **`void`** specifies no conversion in the sense that the value is discarded. As footnote 87 points out, it is possible to cast an operand to the type it already has and change its value.

1142 footnote 87

The footnote was moved to normative text by the response to DR #318.

C++

The C++ Standard explicitly permits an expression to be cast to its own type (5.2.11p1), but does not list any exceptions for such an operation.

Common Implementations

The special case of a cast that specifies no conversion is likely to be subsumed into an implementation's general handling of machine code generation for cast operations.

Coding Guidelines

Technically, a cast that specifies no conversion is a redundant operation. A cast that has no effect can occur for a number of reasons:

190 redundant code

- When typedef names are used; for instance, converting a value having type `TYPE_A` to `TYPE_B`, when both typedef names are defined to have the type **`int`**.
- When an object is defined to have different types in different arms of a **`#if`** preprocessing directive. Casts of such an object may be redundant when one arm is selected, but not the other.
- When an argument to a macro invocation is cast to its declared type in the body of the macro.
- When an invocation of a macro is cast to the type of the expanded macro body.
- Developer incompetence, or changes to existing code, such that uses no longer fit in the above categories.

Some redundant casts may unnecessarily increase the cost of comprehending source code. (Their existence in source code will require effort to process; a redundant cast may generate additional effort because it is surprising and the reader may invest additional effort investigating it.) However, in practice many redundant casts exist for a good reason. But, providing a definition for an *unnecessary* redundant cast is likely to be a complex task. A guideline recommending against some form of redundant casts does not appear to be worthwhile.

While a cast operation may not specify any conversion based on the requirements contained in the C Standard, it may specify a conceptual conversion based on the representation of the application domain (i.e., the types of the cast and its operand are specified using different typedef names which happen to use the same underlying type). Checking that the rules behind these conceptual conversions are followed requires support from a static analysis tool (at the translator level these conversions appear to be redundant code).

redundant code 190

Example

```
1  typedef int TYPE_A;
2  typedef int TYPE_B;
3
4  extern signed int i_1;
5  extern TYPE_A t_1;
6
7  #include "stuff.h"
8
9  void f(void)
10 {
11 int loc_1 = i_1;
12 int loc_2 = (int)i_1;
13 int loc_3 = (TYPE_B)i_1;
14 int loc_4 = t_1;
15 int loc_5 = COMPLEX_EXPR;
16 int loc_6 = (int)COMPLEX_EXPR;
17 int loc_7 = (int)loc_1;
18
19 TYPE_B toc_1 = t_1;
20 TYPE_B toc_2 = (int)t_1;
21 TYPE_B toc_3 = (TYPE_B)t_1;
22 TYPE_B toc_4 = i_1;
23 TYPE_B toc_5 = COMPLEX_EXPR;
24 TYPE_B toc_6 = (TYPE_B)COMPLEX_EXPR;
25 TYPE_B toc_7 = (TYPE_B)toc_1;
26 }
```

Forward references: equality operators (6.5.9), function declarators (including prototypes) (6.7.5.3), simple assignment (6.5.16.1), type names (6.7.6). 1139

footnote 86

86) A cast does not yield an lvalue. 1140

Commentary

The idea behind the cast operator is to convert values, not the types of objects. A cast of a value having a point type may not be an lvalue, but the result can be dereferenced to yield an lvalue.

```
1  extern int *p_i;
2
3  void f(void)
4  {
5  (int)*p_i = 3;   /* Constraint violation, left operand not an lvalue.      */
```

```
6   *(int *)p_i = 3; /* Does not affect the conformance status of the program. */
7   }
```

C++

The result is an lvalue if T is a reference type, otherwise the result is an rvalue. 5.4p1

Reference types are not available in C, so this specification is not a difference in behavior for a conforming C program.

Common Implementations

Some implementations (e.g., `gcc`) allow casts to yield an lvalue.

```
1   int loc;
2
3   (char)loc = 0x02; /* Set one of the bytes of an object to 0x02. */
```

1141 Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type.

Commentary

Type qualifiers affect how translators treat objects, not values. The standard specifies some requirements on pointers to unqualified/qualified versions of types.

1478 qualifier meaningful for lvalues
559 pointer to qualified/unqualified types

C++

Casts that involve qualified types can be a lot more complex in C++ (5.2.11). There is a specific C++ cast notation for dealing with this form of type conversion, `const_cast<T>` (where T is some type).

[Note: some conversions which involve only changes in cv-qualification cannot be done using ***`const_cast`****. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior.* 5.2.11p12

The other forms of conversions involve types not available in C.

Coding Guidelines

Casting to a qualified type can occur through the use of typedef names. Explicitly specifying a type qualifier in the visible source is sufficiently rare that it does not warrant a guideline (while it may not affect the behavior of a translator, the developer's beliefs about such a cast are uncertain).

---

footnote 87

1142 ~~87)~~ If the value of the expression is represented with greater precision or range than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type.

Commentary

This footnote clarifies that the permission to perform operations with greater precision (values having floating-point types may be represented to greater precision than is implied by the type of the expression) does not apply to the cast operator. The issues surrounding this usage are discussed elsewhere.

695 float promoted to double or long double
354 FLT_EVAL_ME

The footnote was moved to normative text by the response to DR #318.

C++

The C++ Standard is silent on this subject.

Common Implementations

The representation used during the evaluation of operands having floating-point type is usually dictated by the architecture of the processor. Processors that operate on a single representation often have instructions for converting to other representations (which can be used to implement the cast operation). An alternative implementation technique, in those cases where no conversion instruction is available, is for the implementation to specify all floating-point types as having the same representation.

6.5.5 Multiplicative operators

multiplicative-expression syntax


```
multiplicative-expression:
                cast-expression
                multiplicative-expression * cast-expression
                multiplicative-expression / cast-expression
                multiplicative-expression % cast-expression
```

Commentary

The use of the term *multiplicative* is based on the most commonly occurring of the three operators.

Table 338.2 lists various results from operating on infinities and NaNs; annex F (of the C Standard) discusses some of the expression optimizing transformations these results may prohibit.

C++

In C++ there are two operators (pointer-to-member operators, `.*` and `->*`) that form an additional precedence level between *cast-expression* and *multiplicative-expression*. The nonterminal name for such expressions is *pm-expression*, which appears in the syntax for *multiplicative-expression*.

Other Languages

Many languages designed before C did not support a remainder operator. Pascal uses **rem** to denote this operator, while Ada uses both **rem** and **mod** (corresponding to the two interpretation of its behavior).

Common Implementations

Multiplicative operators often occur in the bodies of loops with one of their operands being a loop counter. It is sometimes possible to transform the loop, by making use of a regular pattern of loop increments; multiplicative operations can be replaced by additive operations,[277] while replacing division, remainder, or modulo operations requires the introduction of a nested loop.[1223]

Coding Guidelines

expression 943.1 shall be parenthesized

No deviation is listed in the parenthesizing guideline recommendation for the remainder operator because developers are very unlikely to have received any significant practice in its usage (compared to the other two operators, which will have been encountered frequently during their schooling).

Unlike multiplication, people do not usually learn division tables at school. The published studies of multiplicative operators have not been as broad and detailed as those for the additive operators. Many of those that have been performed have investigated mental multiplication, with a few studies of mental division (those that have been performed involved operands that gave an exact integer result, unlike division operations in source code which may not be exact), and your author could find none investigating the remainder operation.

- A study by Parkman[1056] showed subjects a series of simple multiplication problems of the form $p \times q = n$ and asked them to respond as quickly as possible as to whether the value n was correct (both p and q were single-digit values). The time taken for subjects to respond was measured. The results followed the pattern found in an earlier study involving addition; that is, it was possible to fit the results to a number of straight lines (i.e., $RT = ax + b$). In one case x was determined by the value $\min(p, q)$, in the other by $p * q$.

additive-expression syntax 1153

- A study by LeFevre, Bisanz, Daley, Buffone, Greenham, and Sadesky[825] gave subjects single-digit multiplication problems to solve and then asked them what procedure they had used to solve the problems. In 80% of trials subjects reported retrieving the answer from memory; other common solution techniques included: 6.4% deriving it (e.g., $6\times7 \Rightarrow 6\times6+1$), 4.5% number series (e.g., $3\times5 \Rightarrow 5, 10, 15$), and 3.8% repeated addition (e.g., $2\times4 \Rightarrow 2+2$).
- A study by Campbell[194] measured the time taken for subjects to multiply numbers between two and nine, and to divide a number (that gave an exact result) by a number between two and nine. The results showed a number of performance parallels (i.e., plots of their response times and error rates, shown in Figure 1143.1, had a similar shape) between the two operations, although division was more difficult. These parallels suggest that similar internal processes are used to perform both operations.
- A study by LeFevre and Morris[826] supported the idea that multiplication and division are stored in separate mental representations and that sometimes the solution to difficult division problems was recast as a multiplication problem (e.g., $56/8$ as $8\times? = 56$).

Adults (who spent five to six years as children learning their times tables) can recall answers to single digit multiplication questions in under a second, with an error rate of 7.6%.[195] The types of errors made can be put into the following categories[543] (see Table 1143.1):

- *Operand errors.* One of the digits in an operand being multiplied is replaced by another digit. For example, $8\times8 = 40$ is an operand error because it shares an operand, 8, with $5\times8 = 40$.
- *Close operand errors.* This is a subclass of operand errors, with the replaced digit being within ±2 of the actual digit (e.g., $5\times4 = 24$). This behavior is referred to as the *operand distance effect*.
- *Frequent product errors.* The result of the multiplication is given as one of the numbers 12, 16, 18, 24 or 36. These five numbers frequently occur as the result of multiplying operands between 2 and 9.
- *Table errors.* Here the result is a value that is an answer to multiplying two unrelated digits, than those actually given (e.g., $4\times5 = 12$).
- *Operation errors.* Here the multiplication operation is replaced by an additional operation (e.g., $4\times5 = 9$).
- *Non-table errors.* Here the result is a value that is not an answer to multiplying any two single digits; for instance, $4\times3 = 13$, where 13 is not the product of any pair of integers.

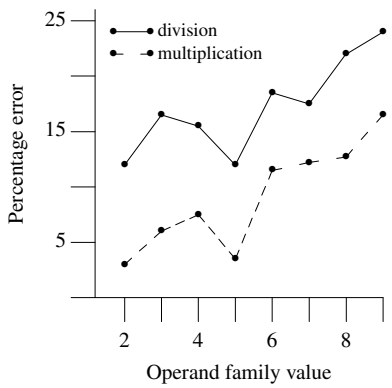


Figure 1143.1: Mean percentage of errors in simple multiplication (e.g., 3×7) and division (e.g., $81/9$) problems as a function of the operand value (see paper for a discussion of the effect of the relative position of the minimum/maximum operand values). Adapted from Campbell.[194]

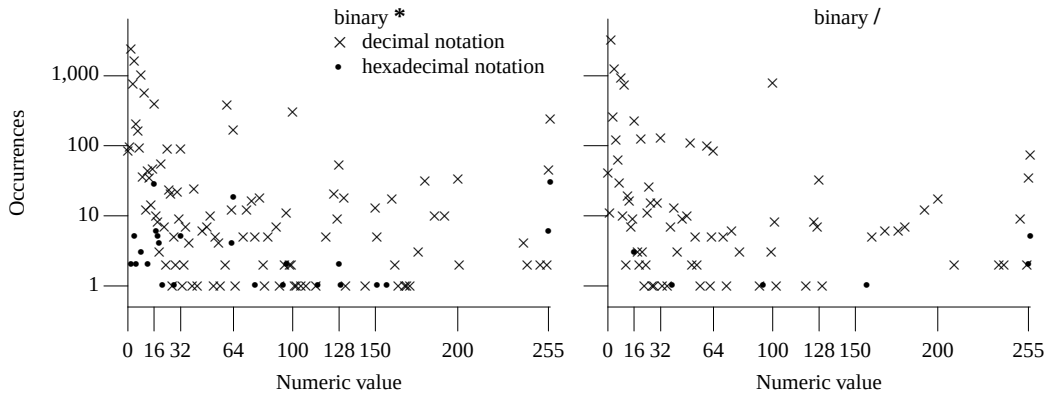


Figure 1143.2: Number of *integer-constant*s having a given value appearing as the right operand of the multiplicative operators. Based on the visible form of the `.c` files.

Table 1143.1: Percentage breakdown of errors in answers to multiplication problems. Figures are mean values for 60 adults tested on 2×2 to 9×9 from Campbell and Graham,[195] and 42 adults tested on 0×0 to 9×9 from Harley.[543] For the Campbell and Graham data, the operand error and operation error percentages are an approximation due to incomplete data.

	Campbell and Graham	Harley
Operand errors	79.1	86.2
Close operand errors	76.8	76.74
Frequent product errors	24.2	23.26
Table errors	13.5	13.8
Operation error	1.7	13.72
Error frequency	7.65	6.3

For a discussion of multiplication of operands containing more than one digit see Dallaway.[310]

Example

See annex G.5.1 EXAMPLE 1 and EXAMPLE 2 for an implementation of complex multiple and divide functions.

Table 1143.2: Common token pairs involving multiplicative operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: a consequence of the method used to perform the counts is that occurrences of the sequence *identifier* `*` are over estimated (e.g., occurrences of a typedef name followed by a `*` are included in the counts).

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier `*`	3.4	92.1	`/ sizeof`	9.0	3.6
identifier `%`	0.0	57.7	`*` identifier	76.5	2.8
identifier `/`	0.1	54.3	`* )`	14.4	2.0
`) /`	0.3	33.9	*floating-constant* `/`	5.8	1.8
`) %`	0.1	31.8	`/` *integer-constant*	53.5	0.5
`*` *floating-constant*	0.2	12.5	`%` *integer-constant*	44.8	0.1
`* sizeof`	1.6	11.2	`/` identifier	27.5	0.1
integer-constant `/`	0.1	8.5	*floating-constant* `*`	6.8	0.1
`, %`	0.0	6.5	`/ (`	7.9	0.1
`/` *floating-constant*	2.0	6.4	`%` identifier	47.6	0.0
`* *v`	1.4	4.4			

Constraints

multiplicative operand type

Each of the operands shall have arithmetic type. 1144

Commentary

These operators have no common usage for anything other than arithmetic types.

Coding Guidelines

Enumerated types are often thought about in symbolic, not arithmetic, terms. The use of operands having an enumerated type is discussed elsewhere.

822 symbolic name
517 enumeration set of named constants

A boolean value may be thought about in terms of a zero/nonzero representation. In this case a multiplication operation involving an operand representing a boolean value will return a boolean (in the zero/nonzero sense) result. In this case multiplication is effectively a logical-OR operation. Can a parallel be drawn between such usage and using bitwise operations to perform arithmetic operations (with the aim of using similar rationales to make guideline recommendations)? Although this might be an interesting question, given the relative rareness of the usage these coding guidelines do not discuss the issue further.

476 boolean role

Table 1144.1: Occurrence of multiplicative operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	%	_int	40.6	_unsigned long	*	_int	2.8
int	/	_int	25.6	int	/	float	2.7
other-types	*	other-types	18.1	long	/	_int	2.5
other-types	/	other-types	16.2	unsigned long	%	int	2.3
_int	*	_int	13.4	_int	*	unsigned short	2.2
unsigned int	%	_int	12.6	_int	*	_unsigned long	2.2
int	%	int	12.2	_unsigned long	*	int	2.1
int	*	_int	12.1	unsigned long	*	_unsigned long	1.9
_int	/	_int	11.0	int	%	unsigned int	1.8
_unsigned long	/	_unsigned long	9.9	float	/	float	1.8
_unsigned long	*	unsigned char	9.5	_unsigned long	/	_int	1.6
int	*	_unsigned long	8.8	unsigned int	%	int	1.6
float	*	float	8.8	unsigned long	%	unsigned long	1.5
other-types	%	other-types	7.3	unsigned short	/	_int	1.3
unsigned long	/	_int	6.6	unsigned long	/	unsigned long	1.3
_int	*	int	6.5	unsigned int	*	_int	1.3
int	*	int	5.9	unsigned int	*	_unsigned long	1.2
unsigned long	/	_unsigned long	5.8	int	/	_unsigned long	1.2
unsigned int	/	_int	5.3	_double	/	_double	1.2
int	/	int	5.0	float	*	_int	1.1
unsigned int	%	unsigned int	4.2	unsigned long	*	_int	1.0
int	%	unsigned long	4.2	unsigned int	%	unsigned long	1.0
int	%	_unsigned long	3.9	int	/	unsigned long	1.0
long	%	_int	3.7	_int	*	unsigned int	1.0
unsigned long	%	_int	3.1				

1145 The operands of the % operator shall have integer type.

Commentary

The modulus operator could have been defined to include arithmetic types. However, processors rarely include instructions for performing this operation using floating-point operands.

Semantics

1146 The usual arithmetic conversions are performed on the operands.

Commentary

These conversions may result in the integer promotions being performed.

710 arithmetic conversions integer promotions

Common Implementations

All three operators are sufficiently expensive to implement[1263] (both in terms of transistors needed by the hardware and time taken to execute the instruction) that some processors contain instructions where one or more operands has a type narrower than **int**. A translator can make use of such instructions, invoking the as-if rule, when the operands have the appropriate bit width; for instance, multiplying two 8-bit operands to yield a 16-bit result.

A number of processors do not support integer multiply and/or divide instructions in hardware (e.g., early high-performance processors CDC 6x00 and 7x00, Cray; most low-cost processors; RISC processors seeking to simply their design— HP PA-RISC 1.0, 1.1, 2.0,[571] HP–was DEC– Alpha;[1247] Sun SPARC, pre-version 8[1450]).

A recently proposed hardware acceleration technique[239] is to store the results of previous multiplication and division operations in a cache, reusing rather than recalculating the result whenever possible. (Dynamic profiling has found that a high percentage of these operations share the same operands as previous operations.)

---

binary * result

The result of the binary * operator is the product of the operands. 1147

Commentary

array 992 n-dimensional reference pointer 1167 arithmetic addition result

Source code can contain implicit multiplications as well as explicit ones; for instance, array indexing. If imaginary types are supported, the expression `Infinity*I` is simply a way of obtaining an infinite imaginary value.

Common Implementations

Multiplication is sufficiently common and generally executes sufficiently slowly that implementations often put a lot of effort into alternative implementation strategies. A common special case is multiplying by a known (at translation time) constant.[885, 1443]

A multiply instruction can be expressed as a combination of one or more add, subtract, or left shift (by any amount) instructions. For instance, multiplying by 9 is equivalent to shifting a value left 3 bits and adding in the original value, the later probably executing in a few machine cycles. A general expression giving the minimum number of add/subtract/shift instructions required for any constant is not known, but the following are some upper bounds:

- multiplication by an n-bit constant can always be performed using at most n instructions,[1443]
- if the constant n contains many 0-bits, an algorithm that generates $4g(n) + 2s(n) - 1 - \theta$ instructions, where $g(n)$ is the number of groups of two or more consecutive 1-bits in the constant n, $s(n)$ is the number of single 1-bits, and θ is 1 if n ends in a 1-bit and 0 otherwise, may produce a shorter sequence,
- if a fused shift–add instruction is available (e.g., HP PA-RISC[571]) six or less of these instructions are required to multiply by any constant between 0 and 10,000.[885]

multiply always truncate IEC 60559 29 correctly 64 rounded result

Approximately half of the hardware needed to perform a IEEE compliant floating-point multiply is only there to guarantee a correctly rounded result. Significant savings in power consumption and execution delays can be achieved by not providing this guarantee[1205] and a number of vendors[350, 626, 971] support multiplication instructions that do not guarantee this behavior (e.g., they always truncate).

Usage

Measurements by Citron, Feitelson, and Rudolph[239] found that in a high percentage of cases the operands of multiplication operations repeat themselves (59% for integer operands and 43% for floating-point). Measurements were based on maintaining previous results in a 32-entry, 4-way associative, cache.

---

binary / result

The result of the / operator is the quotient from the division of the first operand by the second; 1148

Commentary

The identity $1/\infty \Rightarrow 0$ is consistent with the possibility that the infinity was the result of a division by zero.

Common Implementations

Because of its iterative nature, a divide operation is expensive in terms of the number of transistors required to implement it (in hardware) and its execution time. Divide instructions are often the slowest operation (involving integer operands) on a processor (it can take up to 60 cycles on the HP–was DEC– Alpha 21064 and 46 cycles on the Intel Pentium). Internally it is often implemented as a sequence of steps, computing the quotient digit by digit using some recursive formula; combinatorial implementations replicate the hardware that performs the division step (up to n times, where n is the number of bits in the significand). Because it is an infrequently used operation processor designers often trade significant amounts of chip resources (transistors) against performance (greater performance requires greater numbers of transistors). Many low-cost processors do not contain a divide instruction. For different reasons most of the early versions of the commercial RISC processors did not include a divide instruction. What was provided were one or more simpler instructions that could be used to create a sequence of code capable of performing a division operation. The length of the sequence is usually proportional to the number of bits in the narrowest operand.

Dynamic profiling information of hydro2d (from the SPEC92)[189] found that 64% of the executed divide instructions had either a 0 for its numerator or a 1 for its denominator. Using value profiling techniques, they were able to reduce the execution time of hydro2d by 15%, running on a HP–was DEC– Alpha 21064 processor; and were able to reduce the number of cycles executed by an Intel Pentium running some games' programs by an estimated 5%.

940 value profiling

An alternative to performing a divide is to take the reciprocal and multiply. This technique was used by Cray for floating-point operands in the hardware implementation of some of their processors.[299] The Diab Data compiler[353] `-Xieee754-pedantic` option enables developers to specify that the convert divide-to-multiple optimization should not be attempted. At least one processor vendor has proposed this optimization for integer operations.[23]

A common optimization is to replace division by a constant divisor that is a power of 2 by the appropriate right shift, when the numerator value is known to be positive. A less well known optimization[1443] enables any division by a constant to be replaced by a multiply. The basic idea is to multiply by a kind of reciprocal of the divisor (this reciprocal value, on a 32-bit processor, has a value close to $2^{32}/d$) and to extract the most significant 32 bits of the product.

Magenheimer, Peters, Pettis, and Zuras[885] describe an algorithm for obtaining sequences of shifts and adds that are equivalent to division by a constant.

Coding Guidelines

Many developers are aware of the relatively poor performance of the divide operator and sometimes transform the expression to use another operator. The issue of a right-shift operator being used to divide by a power of two is discussed elsewhere, as is the issue of using a bitwise operator to perform an arithmetic operation. A divide by a floating constant might be rewritten as a multiply by its reciprocal `x/5.0` becoming `x*0.2`. However, such a transformation does not yield numerically equivalent expressions unless the constants have an exact representation (in the floating-point representation used by the implementation executing the program).

1194 right-shift result
945 bitwise operators

Testing code containing a division operator can require checking a number of special cases. It is possible for the left operand to be incorrect and still obtain the correct answer most of the time. For instance, if `(x+1)/101` had been written instead of `(x+2)/101`, the result would be correct for 99% of the values of `x`. A white-box testing strategy would use test cases that provide left operand values just less than and just greater than the value of the right operand, however, testing is not within the scope of these coding guidelines.

Usage

Measurements by Oberman[1024] found that in a high percentage of cases division operations on floating-point operands repeat themselves (i.e., the same numerator and denominator values). The measurements were done using the SPECfp92 and NAS (Fortran-based) benchmarks.[82] Simulations using an infinite division operand cache found a hit rate (i.e., cache lookup could return the result of the division) of 69.8%, while a cache containing 128 entries had a hit rate of 60.9%. A more detailed analysis by Citron, Feitelson, and

0 SPEC benchmarks

Rudolph[239] found a great deal of variability over different applications, with multimedia applications having hit rates of 50% (using a 32 entry, 4-way associative cache).

% operator result

the result of the % operator is the remainder. 1149

Commentary

The character % is one of the few characters not already used as an operator in other programming languages that appears on most keyboards. It also is also visually similar to the character used to represent the divide operator token.

Common Implementations

In some processors the division instruction also calculates the remainder, putting it in a separate register. While a few processors have instructions that only perform the remainder operation, many more do not support this operation in hardware.

binary / 1148 result

The reciprocal-multiply algorithm listed as a possible replacement for division by a constant can also be used to calculate the remainder when the right operand is constant.

Coding Guidelines

binary / 1148 result

Like the divide operator, the performance of the remainder operator is generally poor and there are a number of well-known special cases that can be optimized. For instance, when the first operand has an unsigned type and the value of the second operand is a power of two, a bitwise-AND operation can zero out the top bits. Replacing an arithmetic operator by equivalent bitwise operations is discussed elsewhere.

bitwise operators 945

The result of this operator, when the left operand is positive, will range from zero to one less than the value of the right operand. Sometimes a value that varies between one and the value of the right operand is required. This can lead to off-by-one mistakes like those commonly seen in array subscripting.

---

divide by zero
remainder by zero

In both operations, if the value of the second operand is zero, the behavior is undefined. 1150

Commentary

The usual mathematical convention (for divide) is to say that the result is infinity. The C integer types do not support a representation for infinity; however, the IEC 60559 representation does. The IEC 60559 Standard defines three possible results, depending on the value of the left operand: $+\infty$, $-\infty$, and NaN.

IEC 60559 29

The behavior can also depend on the setting of the FENV_ACCESS pragma.

Other Languages

Some languages regard a second operand of zero for this operator as being more serious than undefined behavior (e.g., Ada requires an exception to be raised).

Common Implementations

Many processors raise an exception if the second operand is zero and has an integer type. This exception may be catchable within a program via a signal handler.

Coding Guidelines

Having a zero value for the second operand rarely makes any sense, algorithmically. It is unusual for a developer to intend it to occur. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines 0 not faults

When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.[88] 1151

Commentary

C99 reflects almost universal processor behavior (as does the Fortran Standard). This definition truncates toward zero and the expression (-(a/b) == (-a)/b) && (-(a/b) == a/(-b)) is always true. It also means that the absolute value of the result does not depend on the signs of the operands; for example:

```
+10 / +3 == +3      +10 % +3 == +1      -10 / +3 == -3      -10 % +3 == -1

+10 / -3 == -3      +10 % -3 == +1      -10 / -3 == +3      -10 % -3 == -1
```

C90

When integers are divided and the division is inexact, if both operands are positive the result of the / operator is the largest integer less than the algebraic quotient and the result of the % operator is positive. If either operand is negative, whether the result of the / operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the % operator.

If either operand is negative, the behavior may differ between C90 and C99, depending on the implementation-defined behavior of the C90 implementation.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  int x = -1,
6      y = +3;
7
8  if ((x%y > 0) ||
9     ((x+y)%y == x%y))
10    printf("This is a C90 translator behaving differently than C99\n");
11 }
```

Quoting from the C9X Revision Proposal, WG14/N613, that proposed this change:

The origin of this practice seems to have been a desire to map C's division directly to the "natural" behavior of the target instruction set, whatever it may be, without requiring extra code overhead that might be necessary to check for special cases and enforce a particular behavior. However, the argument that Fortran programmers are unpleasantly surprised by this aspect of C and that there would be negligible impact on code efficiency was accepted by WG14, who agreed to require Fortran-like behavior in C99. WG14/N613

C++

If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined[74)]. 5.6p4

Footnote 74 describes what it calls *the preferred algorithm* and points out that this algorithm follows the rules by the Fortran Standard and that C99 is also moving in that direction (work on C99 had not been completed by the time the C++ Standard was published).

The C++ Standard does not list any options for the implementation-defined behavior. The most likely behaviors are those described by the C90 Standard (see C90/C99 difference above).

Other Languages

Ada supports two remainder operators (**rem** and **mod**) corresponding to the two interpretations of the remainder operator (round toward or away from zero when one of the operands is negative).

Common Implementations

All C90 implementations known to your author follow the C99 semantics.

Coding Guidelines

While experience shows that developers do experience difficulties in comprehending code where one or more of the operands of the / operator has a negative value, there are no obvious guideline recommendations.

Example

```
1  int residua_modulo(int x, int y) /* Assumes y != 0 */
2  {
3  if (x >= 0)
4     return x%y;
5  else
6     return x%y + ((y > 0) ? +y : -y);
7  }
```

If the quotient **a/b** is representable, the expression **(a/b)*b + a%b** shall equal **a**. 1152

Commentary

This requirement was in C90 and only needed to be this complex because there were two possibilities for the result of the division operator in C90.

6.5.6 Additive operators

---

additive-expression syntax additive operators 1153

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

Commentary

The use of the term *additive* is based on the most commonly occurring of the two operators.

The probability that the sum of two floating-point numbers, randomly chosen from a logarithmic distribution, overflows is given by[415] (the probability of the subtraction underflowing has a sightly more complicated form, however it differs by at most 1 part in 10^{-9} of the probability given by this formula):

$$\frac{\pi^2}{6(\ln(\Omega/\omega))^2} \tag{1153.1}$$

where Ω and ω are the largest and smallest representable values respectively.

EXAMPLE 381 IEC 60559 floating-point

In the case of a representation meeting the minimum requirements of IEC 60559 the overflow probabilities are 7×10^{-7} for single-precision and 4×10^{-11} for double-precision. In practice application constraints may significantly limit the likely range of operand values (see Table 334.1), resulting in a much lower probability of overflow.

Table 338.2 lists various results from operating on infinities and NaNs; annex F (of the C Standard) discusses some of the expression optimizing transformations these results may prohibit.

Other Languages

These operators have the same precedence in nearly all languages. Their precedence relative to the multiplicative operators is also common to nearly all languages and with common mathematical usage.

Coding Guidelines

A study by Parkman and Groen[1057] showed subjects a series of simple addition problems of the form $p + q = n$ and asked them to respond as quickly as possible as to whether the value n was correct (both p and q were single-digit values). The time taken for subjects to respond was measured. It was possible to fit the results to a number of straight lines (i.e., $RT = ax + b$). In one case x was determined by the value $\min(p, q)$ in the other by $p + q$. That is, the response time increased as the minimum of the two operands increased, and as the sum of the operands increased. In both cases the difference in response time between when n was correct and when it was not correct (i.e., the value of b) was approximately 75 ms greater for incorrect. The difference in response time over the range of operand values was approximately 175 ms.

0 rule-base mistakes

VanLehn studied the subtraction mistakes made by school-children. The results showed a large number of different kinds of mistakes, with no one mistake being significantly more common than the rest (unlike the situation for multiplication).

1147 binary * result

The so-called *problem size* effect describes the commonly found subject performance difference between solving arithmetic problems involving large numbers and small numbers (e.g., subjects are slower to solve $9 + 7$ than $2 + 3$). A study by Zbrodoff[1502] suggested that this effect may be partly a consequence of the fact that people have to solve arithmetic problems involving small numbers more frequently than larger numbers (i.e., it is a learning effect); interference from answers to other problems was also a factor.

Table 1153.1: Common token pairs involving additive operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier +	1.0	77.5	+ **sizeof**	1.5	3.8
identifier -	0.5	75.7	+ *integer-constant*	33.7	1.9
) -	0.3	14.7	- *integer-constant*	44.0	1.3
) +	0.6	12.9	+ identifier	55.4	0.7
+ *floating-constant*	0.4	7.7	+ (	8.3	0.4
integer-constant +	0.4	6.3	- identifier	46.1	0.3
integer-constant -	0.2	5.8	- (	6.2	0.1

Constraints

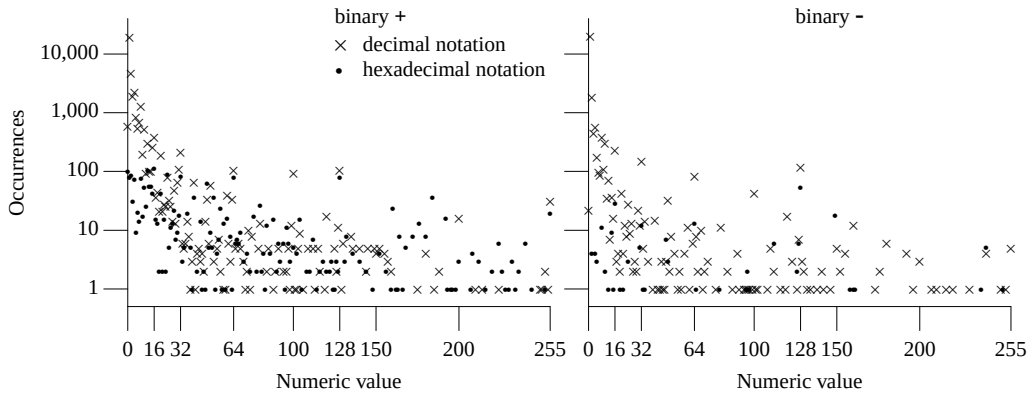


Figure 1153.1: Number of *integer-constant*s having a given value appearing as the right operand of additive operators. Based on the visible form of the `.c` files.

addition operand types

For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type. 1154

Commentary

null pointer 748 constant

Both operands in the expression 0+0 have to be interpreted as having an arithmetic type. Although 0 can be interpreted as the null pointer constant, treating it as such would violate this constraint (because it is not a pointer to object type).

C++

The C++ Standard specifies that the null pointer constant has an integer type that evaluates to zero (4.10p1). In C the `NULL` macro might expand to an expression having a pointer type. The expression `NULL+0` is always a null pointer constant in C++, but it may violate this constraint in C. This difference will only affect C source developed using a C++ translator and subsequently translated with a C translator that defines the `NULL` macro to have a pointer type (occurrences of such an expression are also likely to be very rare).

Other Languages

Many languages do not support pointer arithmetic (i.e., adding integer values to pointer values). Some languages use the `+` operator to indicate string concatenation.

Coding Guidelines

array sub-989 script identical to

When some coding guideline documents prohibit the use of pointer arithmetic, further investigation often reveals that the authors only intended to prohibit the operands of an arithmetic operation having a pointer type. As discussed elsewhere, it is not possible to prevent the use of pointer arithmetic by such a simple-minded interpretation of C semantics.

symbolic 822 name enumeration 517 set of named constants postfix 1081 operator operand multiplicative 1144 operand type

Enumerated types are often thought about in symbolic, not arithmetic terms. The use of operands having an enumerated type is discussed elsewhere. The rationale for the deviation given for some operators does not apply to the addition operator.

The possibility of arithmetic operations on operands having a boolean role is even less likely than for the multiplication operators.

Table 1154.1: Occurrence of additive operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	+	**_int**	37.5	**unsigned long**	+	**_int**	2.6
int	-	**_int**	19.5	**unsigned long**	-	**unsigned long**	2.4
other-types	+	other-types	16.2	**unsigned int**	-	**unsigned int**	2.2
other-types	-	other-types	16.0	**long**	-	**_int**	2.2
_int	+	**_int**	11.8	**_int**	-	**int**	2.1
int	-	**int**	10.8	ptr-to	-	**_int**	2.0
_int	-	**_int**	8.8	**long**	-	**long**	2.0
ptr-to	-	ptr-to	6.4	**unsigned int**	+	**_int**	1.7
ptr-to	+	**unsigned long**	6.2	**float**	+	**float**	1.7
ptr-to	+	**long**	5.8	**unsigned short**	-	**int**	1.5
float	-	**float**	5.0	**unsigned long**	+	**unsigned long**	1.4
unsigned long	-	**_int**	4.9	**int**	-	**unsigned short**	1.4
int	+	**int**	4.7	**_int**	+	**int**	1.4
unsigned int	-	**_int**	4.2	**unsigned short**	+	**_int**	1.2
ptr-to	+	**int**	3.7	**unsigned short**	-	**_int**	1.1
_unsigned long	-	**_int**	3.1	**unsigned char**	-	**_int**	1.1
ptr-to	-	**unsigned long**	3.1	**unsigned int**	+	**unsigned int**	1.0
ptr-to	+	**_int**	3.0				

(Incrementing is equivalent to adding 1.) 1155

Commentary

This specifies how the term *incrementing* is to be interpreted in the context of the additive operators.

Coding Guidelines

A common beginner programming mistake is to believe that incrementing an object used for input will cause the next value to be input.

1156 For subtraction, one of the following shall hold:

Commentary

Unlike addition, subtraction is not a symmetrical operator, and it is simpler to specify the different cases via bullet points.

1157 88) This is often called "truncation toward zero". footnote 88

Commentary

Either the term *truncation toward zero* or *rounded toward zero* is commonly used by developers to describe this behavior.

C90

This term was not defined in the C90 Standard because it was not necessarily the behavior, for this operator, performed by an implementation.

C++

Footnote 74 uses the term *rounded toward zero*.

Other Languages

This term is widely used in many languages.

1158 — both operands have arithmetic type;

Commentary

This includes the complex types.

Other Languages

Support for operands of this type is universal to programming languages.

Coding Guidelines

These are the operand types that developers use daily in their nonprogramming lives. The discussion on operands having enumerated types or being treated as boolean, is applicable here.

521 arithmetic type
1154 addition operand types

1159 — both operands are pointers to qualified or unqualified versions of compatible object types; or subtraction pointer operands

Commentary

The additive operators are not defined for operands having pointer to function types. The expression 0−0 is covered by the discussion on operand types for addition.

1154 addition operand types

Although the behavior is undefined unless the two pointers point into the same object, one of the operands may be a parameter having a qualified type. Hence the permission for the two operands to differ in the qualification of the pointed-to type. Differences in the qualification of pointed-to types is guaranteed not to affect the equality status of two pointer values.

1173 pointer subtraction point at same object
746 pointer converting qualified/unqualified

C++

5.7p2

— both operands are pointers to cv-qualified or cv-unqualified versions of the same completely defined object type; or

Requiring the same type means that a C++ translator is likely to issue a diagnostic if an attempt is made to subtract a pointer to an enumerated type from its compatible integer type (or vice versa). The behavior is undefined in C if the pointers don't point at the same object.

```
#include <stddef.h>

enum e_tag {E1, E2, E3}; /* Assume compatible type is int. */
union {
      enum e_tag m_1[5];
      int m_2[10];
      } glob;

extern enum e_tag *p_e;
extern int *p_i;

void f(void)
{
ptrdiff_t loc = p_i-p_e; /* does not affect the conformance status of the program */
                         // ill-formed
}
```

addition 1154 operand types The expression `NULL-0` is covered by the discussion on operand types for addition.

Other Languages

Support for subtracting two values having pointer types is unique to C (and C++).

Table 1159.1: Occurrence of operands of the subtraction operator having a pointer type (as a percentage of all occurrences of this operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
char *	-	**char** *	48.9	**void** *	-	**void** *	1.4
unsigned char *	-	**unsigned char** *	26.2	**int** *	-	**int** *	1.4
struct *	-	**struct** *	13.7	**unsigned short** *	-	**unsigned short** *	1.2
const char *	-	**const char** *	4.6	other-types	-	other-types	0.0

— the left operand is a pointer to an object type and the right operand has integer type. 1160

Commentary

addition 1154 operand types The issues are the same as those discussed for addition.

Table 1160.1: Occurrence of additive operators one of whose operands has a pointer type (as a percentage of all occurrences of each operator with one operand having a pointer type). Based on the translated form of this book's benchmark programs. Note: in the translator used the result of the `sizeof` operator had type `unsigned long`.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
`char *`	-	`unsigned long`	46.0	`unsigned char *`	-	`int`	1.7
`char *`	+	`unsigned long`	27.3	`const char *`	-	`_ int`	1.7
`char *`	+	`long`	26.8	`short *`	-	`_ int`	1.6
other-types	+	other-types	10.6	`char *`	+	`unsigned char`	1.6
`char *`	-	`_ int`	9.5	`char *`	-	`int`	1.6
`struct *`	-	array-index	9.1	`char *`	-	array-index	1.4
`unsigned char *`	-	`_ int`	8.8	`unsigned char *`	+	`unsigned int`	1.3
`unsigned char *`	+	`_ int`	7.4	`unsigned char *`	-	array-index	1.3
`char *`	+	`int`	6.6	`void *`	-	`_ int`	1.2
`unsigned char *`	+	`int`	5.7	`char *`	+	`signed int`	1.2
`struct *`	-	`_ int`	4.7	`unsigned long *`	+	`int`	1.1
`char *`	+	`_ int`	3.6	`struct *`	+	`_ int`	1.1
`unsigned char *`	-	`_ unsigned long`	2.1	`unsigned char *`	+	`unsigned short`	1.0
`char *`	+	`unsigned int`	1.9	`char *`	+	`unsigned short`	1.0
`struct *`	+	`int`	1.8	other-types	-	other-types	0.0

1161 (Decrementing is equivalent to subtracting 1.)

Commentary

This specifies how the term *decrementing* is to be interpreted in the context of the additive operators.

Semantics

1162 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

additive operators semantics

Commentary

There is no requirement to perform the usual arithmetic conversions if only one of the operands has arithmetic type. The only possible other operand type is a pointer and the result has its type. Performing the usual arithmetic conversions may result in the integer promotions being performed.

706 usual arithmetic conversions

710 arithmetic conversions integer promotions

Common Implementations

Most processors use the same instructions for performing arithmetic involving pointer types as they use for arithmetic types. An operand having an arithmetic type is likely to undergo the same conversions, irrespective of the type of the other operand (and the same optimizations are likely to be applicable).

Coding Guidelines

There is a common incorrect assumption that if the operands of the additive operators have the same type, the operation will be performed using that type. This issue is discussed elsewhere. If the guideline recommendation specifying the use of a single integer type is followed these conversions will have no effect.

653 operand convert automatically

480.1 object int type only

1163 The result of the binary + operator is the sum of the operands.

Commentary

It is quite common for a series of values to be added together to create a total. When the operands have a floating type, each addition operation introduces an error. Are some algorithms better than others at minimizing the error in the final result? The following analysis is based on Robertazzi and Schwartz.[1171] This analysis assumes that the relative error for each addition operation will be independent of the others (i.e., have a zero mean) with a variance (mean square error) of σ^2. Let N represent the number of values being added, which are assumed to be positive. These values can have a number of distributions. In a uniform distribution they are approximately evenly distributed between the minimum and maximum values; while in an exponential distribution they are distributed closer to the maximum value.

A series of values can be added to form a sum in a number of different ways. It is known[1466] that the minimum error in the result occurs if the values are added in order of increasing magnitude. In practice most programs loop over the elements of an array, summing them (i.e., the ordering of the magnitude of the values is random). The mean square error for various simple methods of summing a list of values is shown in columns 2 to 6 of Table 1163.1.

Table 1163.1: Mean square error in the result of summing, using five different algorithms, N values having a uniform or exponential distribution; where μ is the mean of the N values and σ^2 is the mean square error that occurs when two numbers are added.

Distribution	Increasing Order	Random Order	Decreasing Order	Insertion	Adjacency
Uniform $(0, 2\mu)$	$0.2\mu^2N^3\sigma^2$	$0.33\mu^2N^3\sigma^2$	$0.53\mu^2N^3\sigma^2$	$2.6\mu^2N^2\sigma^2$	$2.7\mu^2N^2\sigma^2$
Exponential (μ)	$0.13\mu^2N^3\sigma^2$	$0.33\mu^2N^3\sigma^2$	$0.63\mu^2N^3\sigma^2$	$2.63\mu^2N^2\sigma^2$	$4.0\mu^2N^2\sigma^2$

The simple algorithms maintain a single intermediate sum, to which all values are added in turn. Using the observation that the error in addition is minimized if values of similar magnitude are used,[888] more accurate algorithms are possible. An insertion adder takes the result of the first addition and inserts it into the remaining list of vales to be added. The next two lowest values are added and their result inserted into the remaining list, and so on until a single value, the final result, remains. The adjacency algorithm orders the values by magnitude and then pairs them off. The smallest value paired with the next smallest and so on, up to the largest value. Each pair is added to form a new list of values and the process repeated. Eventually a single value, the final result, remains. The error analysis for these two algorithms shows (last two columns of Table 1163.1) a marked improvement. The difference between these sets of results is that the latter varies as the square of the number of values being added, while the former varies as the cube of the number of values.

These formula can be expressed in terms of information content by defining the signal-to-noise ratio as the square of the final sum divided by the mean square error of this sum, then for the simple ordering algorithm the ratio decreases linearly with N. For the insertion and adjacency addition algorithms, it is independent of N. Another way to decrease the error in the simple ordering algorithms is to increase the number of bits in the significand of the result. Performance can be improved to that of the insertion or adjacency algorithm by using $\log_2(sqrt(N))$ additional bits.

As an example, after adding 4,096 values, the mean square error in the result is approximately 1×10^{-2} using the simple ordering algorithms, and approximately 3×10^{-5} for the insertion and adjacency algorithms.

When negative, as well as positive, values are being summed, there are advantages to working from the largest to the smallest. For an analysis of the performance of seven different algorithms, see Mizukami.[948]

Demmel and Hida

The preceding analysis provides a formula for the average error, what about the maximum error? Demmel and Hida[344] analyzed the maximum error in summing a sorted list of values. They proved an upper bound on the error (slightly greater than 1.5 ULPs) provided the condition $N \leq M$ was met, where N is the number of values in the list and $M = 1 + floor(2^{F-f}/(1-2^{-f}))$, f is the precision of the values being added, and F is the precision of the temporary used to hold the running total. They also showed that if $N \geq M + 2$, the relative error could be greater than one (i.e., no relative accuracy).

ULP 346
precision 335
floating-point

Using IEC 60559 representation for single- and double-precision values: if 65,537, single-precision values are summed, storing the running total in a temporary having double-precision; the maximum error will only be slightly greater than 1.5 ULPs. If the temporary has single-precision, then three additions are sufficient for the error to be significantly greater than this.

Other Languages

Some languages also use the binary **+** operator to indicate string concatenation (not just literals, but values of objects).

Common Implementations

Error analysis on the Cray

Some Cray processors do not implement IEC 60559 arithmetic. On the Cray the error analysis formula

for addition and subtraction differs from that discussed elsewhere. Provided $fl(a \pm b)$ does not overflow or underflow, then: 346 error analysis

$$fl(a \pm b) = ((1 + \epsilon_1) \times a) \pm ((1 + \epsilon_2) \times b) \tag{1163.1}$$

In particular, this means that if a and b are nearly equal, so $a - b$ is much smaller than either a or b (known as *extreme cancellation*), then the relative error may be quite large. For instance, when subtracting $1 - x$, where x is the next smaller floating-point number than 1, the answer is twice as large on a Cray C90, and 0 when it should be nonzero on a Cray 2. This happens because when x's significand is shifted to line its binary point up with 1's binary point, prior to actually subtracting the significands, any bits shifted past the least significant bit of 1 are simply thrown away (differently on a Cray C90 and Cray 2) rather than participating in the subtraction. On IEC 60559 machines, there is a so-called guard digit (actually 3 of them) to store any bits shifted this way, so they can participate in the subtraction.

Usage

A study by Sweeney[1323] dynamically analyzed the floating-point operands of the addition operator. In 26% of cases the values of the two operands were within a factor of 2 of each other, in 13% of cases within a factor of 4, and in 84% of cases within a factor of 1,024.

1164 The result of the binary - operator is the difference resulting from the subtraction of the second operand from the first. subtraction result of

Commentary

If the floating-point operands of a subtraction operator differ from each other by less than a factor of 2, and the result is correctly rounded, then the result is exact[500] (mathematically, $x/2 < y < 2x \Rightarrow x \ominus y = x - y$). 64 correctly rounded result

Common Implementations

Some implementations of this operator complement the value of the right operand and perform an addition. For IEC 60559 arithmetic the expression 0-0 yields -0 when the rounding direction is toward -∞. Cuyt and Verdonk[306] describe the very different results obtained, using two different processors with various combinations of floating-point types, when evaluating an expression that involves subtracting operands having very similar values.

Example

```
1  #include <stdio.h>
2
3  extern float ef1, ef2;
4
5  void f(void)
6  {
7  if (((ef1 - ef2) == 0.0) &&
8      (ef1 != ef2))
9     printf("ef1 and ef2 have very small values\n");
10 }
```

1165 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. additive operators pointer to object

Commentary

Pointer types specify the type they point at; nothing is assumed about how many objects of that type may be contiguous in storage at the pointed-to location. Common developer terminology is to refer to each of these *elements* as an *object*. The same sentence appears elsewhere in the standard.

1203 relational operators pointer to object
1232 equality operators pointer to object

Other Languages

More strongly typed languages require that pointer declarations fully specify the type of object pointed at. A pointer to **int** is assumed to point at a single instance of that type. A pointer to an object having an array type requires a pointer to that array type. Needless to say these strongly typed languages do not support the use of pointer arithmetic.

Common Implementations

Implementations that do not treat storage as a linear array of some type are very rare. The Java virtual machine is one such. Here the intent is driven by security issues; programs should not be able to access protected data or to gain complete control of the processor. All data is typed and references (Java does not have pointers as such) to allocated storage must access it using the type with which it was originally allocated.

Coding Guidelines

Having a pointer to an object behaves the same as if it pointed to the first element of an array; it is part of the C model of pointer handling. It fits in with behavior specified in other parts of the standard, and modifying this single behavior creates a disjoint pointer model (not a more strongly typed model).

---

pointer arithmetic type

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. 1166

Commentary

The additive operation returns a modified pointer value. This pointer value denotes a location having the pointer's pointed-to type.

---

pointer arithmetic addition result

If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. 1167

Commentary

This specification allows pointers to be treated, by developers, like array indexes. This would not be possible if adding/subtracting integer values to/from pointer values operated at the byte level (unless the operand had pointer to character type).

Common Implementations

byte 54 address unique

pointer 1174 arithmetic subtraction result binary * 1147 result

Because every byte of an object has a unique address, adding one to a pointer to **int**, for instance, requires adding `(1*sizeof(int))` to the actual pointer value. Adding, or subtracting, `x` to a pointer to `type` requires that the pointer value be offset by `(x*sizeof(type))`. Subtracting two pointers involves a division operation. Scalar type sizes are often a power of two and optimization techniques, described elsewhere, can be applied. Structure types, may require an actual divide.

In other words, if the expression **P** points to the *i*-th element of an array object, the expressions **(P)+N** (equivalently, **N+(P)**) and **(P)-N** (where **N** has the value *n*) point to, respectively, the *i+n*-th and *i-n*-th elements of the array object, provided they exist. 1168

Commentary

array sub-script identical to 989

Adding to a pointer value can be compared to adding to an array index variable. The storage, or array, is not accessed, until the dereference operator is applied to it.

---

pointer one past end of object

Moreover, if the expression **P** points to the last element of an array object, the expression **(P)+1** points one past the last element of the array object, and if the expression **Q** points one past the last element of an array object, the expression **(Q)-1** points to the last element of the array object. 1169

Commentary

The concept of *one past the last object* has a special meaning in C. When array objects are indexed in a loop (for the purposes of this discussion the array index may be an object having pointer type rather than the form `a[i]`), it is common practice to start at the lowest index and work up. This often means that after the last iteration of the loop to the array the index has a value that is one greater than the number of elements it contains (however, the array object is never indexed with this value). This common practice makes one past the end of the array object special, rather than the one before the start of the array object (which has no special rules associated with it).

This specification requires that a pointer be able to point one past the end of an object, but it does not permit storage to be accessed using such a value. There is an implied requirement that all pointers to one past the end of the same object compare equal.

The response to DR #221 pointed out that:

Simply put, 10 == 9+1. Based on the "as-if" rule, there is no semantic distinction among any of the following: DR #221

```
v+9+1
(v+9)+1
v+(9+1)
v+10
```

This means that there are many ways of creating a pointer to *one past the last element*, other than adding one to a pointer-to the last element.

1170 If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; pointer arithmetic defined if same object

Commentary

This is a requirement on the implementation. *Overflow* is not a term usually associated with pointer values. Here it is used in the same sense as that used for arithmetic values (i.e., denoting a value that is outside of the accepted bounds).

Common Implementations

The standard does not impose any representation requirements on the value of "one past the last element of an array object". There is a practical problem on host processors that use a segmented memory architecture. If an object occupies an entire segment, there is a unique location for the one past the end pointer, to point. *One past the end* could be represented by adding one to the current pointer value (which for a multiple byte element type would not be a pointer to the next element). 590 pointer segmented architecture

Example

```
1  int a[10];
2
3  void f(void)
4  {
5  int *loc_p1 = (a + 10) - 9; /* Related discussion in DR #221. */
6
7  /*
8   * (a+20) may produce an overflow, any subsequent
9   * operation on the value returned is irrelevant.
10  */
11 int *loc_p2 = (a + 20) - 19 ;
12 int *loc_p3 =  a + 20  - 19 ; /* Equivalent to line above. */
13 int *loc_p4 =  a +(20  - 19); /* Defined behavior.         */
14 }
```

pointer arithmetic undefined

otherwise, the behavior is undefined. 1171

Commentary

Any pointer arithmetic that takes a pointer outside of the pointed-to object (apart from the one past exception) is undefined behavior. There is no requirement that the pointer be dereferenced; creating it within a subexpression is sufficient.

Common Implementations

pointer 590 segmented architecture Motorola 39 56000

Most implementations treat storage as a contiguous sequence of bytes. Incrementing a pointer simply causes it to point at different locations. On a segmented architecture the value usually wraps from either end of the segment to point at the other end. Some processors[6] support circular buffers by using modulo arithmetic for operations on pointer values.

Some implementations perform checks on the results of pointer arithmetic operations during program execution.[681,690] (Most implementations that perform pointer checking only perform checks on accesses to storage.)

Coding Guidelines

Expressions whose intermediate results fall outside the defined bounds that can be pointed out (such as the `(a+20)-19` example above), but whose final result is within those bounds, do occur in practice. In practice, the behavior of the majority of processors results in the expected final value of the expression being returned. Given the behavior of existing processors and the difficulty of enforcing any guideline recommendation (the operands are rarely translation-time constants, meaning the check could only be made during program execution), no recommendation is made here.

one past the end accessing

If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated. 1172

Commentary

&* 1092

Accessing storage via such a result has undefined behavior. The special case of `&*p` is discussed elsewhere.

C++

This requirement is not explicitly specified in the C++ Standard.

Common Implementations

This is one of the checks that has to be performed by runtime checking tools that claim to do pointer checking.[65,548,681,690,1032,1288] The granularity of checking actually performed varies between these tools. Some tools try to minimize the runtime overhead by only performing a small number of checks (e.g., does the pointer point within the stack or within the storage area currently occupied by allocated storage). A few tools check that storage is allocated for objects at the referenced location. Even fewer tools detect a dereference of a *one past the last object* pointer value.

Coding Guidelines

guidelines 0 not faults

Developers rarely intend to reference storage via such a pointer value. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

pointer subtraction point at same object

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; 1173

Commentary

structure members 1206 later compare later

The standard does not contain any requirements on the relative positions, in storage, of different objects. (Although there is a relational requirement for the members of structure types.) Subtracting two pointers that do not point at elements of the same object (or one past the last element) is undefined behavior. This requirement renders `P-Q` undefined behavior when both operands have a value that is equal to the null pointer constant.

C90

The C90 Standard did not include the wording "or one past the last element of the array object;". However, all implementations known to your author handled this case according to the C99 specification. Therefore, it is not listed as a difference.

Common Implementations

Processors do not usually contain special arithmetic instructions for operands having pointer types. The same instruction used for subtracting two integer values is usually used for subtracting two pointer values. This means that checking pointer operands, to ensure that they both point to elements of the same object, incurs a significant runtime time-performance overhead. Only a few of the tools that perform some kind of runtime pointer checking perform this check.[65, 681, 690, 1288] The behavior of most implementations is to treat all storage as a single array object of the appropriate type. Whether the two pointers refer to declared objects does not usually affect behavior.

Coding Guidelines

Developers sometimes do intend to subtract pointers to different array objects and such usage makes use of information on the layout of objects in storage, which is discussed elsewhere.

1354 storage layout

Example

```
#include <stddef.h>

extern void zero_storage(char *, size_t);

void f(void)
{
char start;
/* ... */
char end;

if (&start < &end)
   zero_storage(&start, &end-&start);
else
   zero_storage(&end, &start-&end);
}
```

1174 the result is the difference of the subscripts of the two array elements.

pointer arithmetic subtraction result

Commentary

This specification matches the behavior for adding/subtracting integer values to/from pointer values. The calculated difference could be added to the value of the right operand to yield the value of the left operand.

1167 pointer arithmetic addition result

Common Implementations

Because every byte of an object has a unique address, subtracting two pointers to **`int`**, for instance, requires dividing the result of the subtraction by `sizeof(int)`. Subtracting two pointers to type requires that the subtracted value be divided by `sizeof(type)`.

54 byte address unique

The size of scalar types is often a power of two and an obvious optimization technique is to map the divide to a shift instruction. However, even if the size of the element type is not a power of two, a divide may not need to be performed. A special case of the technique that allows a divide to be replaced by a multiply instruction applies when it is known that the division is exact (i.e., zero remainder).[1443] For instance, on a 32 bit processor, division by 7 can be mapped to multiplication by 0xB6DB6DB7 and extracting the least significant 32 bits of the result.

1148 binary / result

Note: this technique produces widely inaccurate results if the numerator is not exactly divisible, which can occur if the value of one of the pointers involved in the original subtraction has been modified so that it no longer points to the start of an element.

Example

In the following the first `printf` should yield 10 for both differences. The second `printf` should yield the value `sizeof(int)*10`.

```
#include <stdio.h>

void f(void)
{
char a1[20], *p11 = a1, *p12 = a1+10;
int  a2[20], *p21 = a2, *p22 = a2+10;

printf("a1 pointer difference=%d, a2 pointer difference=%d\n",
                                          (p12 - p11), (p22 - p21));
printf("char * cast a2 pointer difference=%d\n",
                                         ((char *)p22 - (char *)p21));
}
```

---

pointer subtract result type

The size of the result is implementation-defined, and its type (a signed integer type) is **`ptrdiff_t`** defined in the **`<stddef.h>`** header. 1175

Commentary

Here the word *size* is referring to the numeric value of the result. The implementation-defined value will be greater than or equal to the value of the `PTRDIFF_MIN` macro and less than or equal to the value of the `PTRDIFF_MAX` macro. There is no requirement that `SIZE_MAX == PTRDIFF_MAX` or that `ptrdiff_t` be able to represent a difference in pointers into an array object containing the maximum number of bytes representable in the type `size_t`.

sizeof 1127 result type

It is the implementation's responsibility to ensure that the type it uses for `ptrdiff_t` internally is the same as the typedef definition of `ptrdiff_t` in the supplied header, **`<stddef.h>`**. A developer can define a typedef whose name is `ptrdiff_t` (provided the name is unique in its scope). Such a declaration does not affect the type used by a translator as the result type for the subtraction operator when both operands have a pointer type.

Other Languages

Most languages have a single integer type, if they support pointer subtraction, there is often little choice for the result type.

Coding Guidelines

While the type `ptrdiff_t` may cause a change in type of subsequent operations within the containing full expression, the effect is not likely to be significant.

sizeof 1127 result type

If the result is not representable in an object of that type, the behavior is undefined. 1176

Commentary

Most processors have an unsigned address space, represented using the same number of bits as the width of the largest integer type supported in hardware. (Operations on values having pointer and integer types are usually performed using the same instructions.) Given that `ptrdiff_t` is a signed type, an implementation only needs to support the creation of an object occupying more than half the total addressable storage before nonrepresentable results can be obtained. (The object need contain only a single element since a pointer to one past the last element can be the second operand.)

width 626 integer type

C90

As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.

Common Implementations

For the majority of processors that use the same instructions for integer and pointer operands the behavior for nonrepresentable results is likely to be the same in both cases. The IBM AS/400 hardware uses a 16-byte pointer. The ILE C documentation[617] is silent on the issue of this result not being representable.

Coding Guidelines

Programs creating objects that are sufficiently large for this undefined behavior to be a potential issue are sufficiently rare that these coding guidelines say nothing about the issue.

1177 In other words, if the expressions `P` and `Q` point to, respectively, the *i*-th and *j*-th elements of an array object, the expression `(P)-(Q)` has the value *i-j* provided the value fits in an object of type `ptrdiff_t`.

Commentary

This sentence specifies the behavior described in the previous C sentences in more mathematical terms.

1178 Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.[89)]

Commentary

The equivalence between these expressions requires that pointer subtraction produce the same result when one or more of the operands point within the same array object or one past the last element of the same array object.

1179 EXAMPLE Pointer arithmetic is well defined with pointers to variable length array types.

```
{
        int n = 4, m = 3;
        int a[n][m];
        int (*p)[m] = a;          // p == &a[0]
        p += 1;                   // p == &a[1]
        (*p)[2] = 99;             // a[1][2] == 99
        n = p - a;                // n == 1
}
```

If array `a` in the above example were declared to be an array of known constant size, and pointer `p` were declared to be a pointer to an array of the same known constant size (pointing to `a`), the results would be the same.

Commentary

The machine code generated by a translator to perform the pointer arithmetic needs to know the size of the type pointed-to, which in this example will not be a constant value that is known at translation time. 548 known constant size

C90

This example, and support for variable length arrays, is new in C99.

Common Implementations

When the pointed-to type is a variable length array, its size is not known at translation time. A variable size on the second or subsequent subscript prevents optimization of any of the implicit multiplication operations needed when performing pointer arithmetic.

pointer 1167 arithmetic addition result

At the time of this writing your author is not aware of any C based pointer-checking tools that support checking on pointers to variably sized arrays.

Forward references: array declarators (6.7.5.2), common definitions `<stddef.h>` (7.17). 1180

6.5.7 Bitwise shift operators

shift-expression syntax

1181

```
shift-expression:
                additive-expression
                shift-expression << additive-expression
                shift-expression >> additive-expression
```

Commentary

Bit shift instructions are ubiquitous in processor instruction sets. These operators allow writers of C source to access them directly. Many processors also contain bit rotate instructions; however, the definition of these instructions varies (in some cases the rotate includes a bit in the status flags register), and they are not commonly called for in algorithms.

Other Languages

Most languages do not get involved in providing such low level operations, just because such instructions are available on most processors. Java also defines the `>>>` operator.

Table 1181.1: Common token pairs involving the shift operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier >>	0.1	63.9	] <<	0.5	5.3
identifier <<	0.1	37.3	<< *integer-constant*	63.4	0.8
integer-constant <<	0.5	36.1	>> *integer-constant*	79.8	0.7
) >>	0.2	28.0	<< identifier	28.4	0.1
) <<	0.2	20.3	<< (	8.1	0.1
] >>	0.4	6.2	>> identifier	15.9	0.0

Constraints

Each of the operands shall have integer type. 1182

Commentary

This constraint reflects the fact that processors rarely contain instructions for shifting non-integer types (e.g., floating-point types), which in turn reflects the fact that there is no commonly defined semantics for shifting other types.

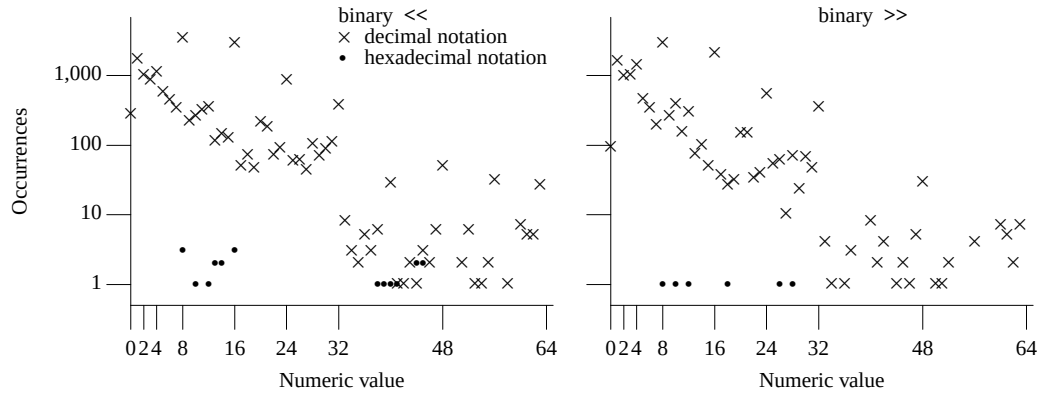


Figure 1181.1: Number of *integer-constant*s having a given value appearing as the right operand of the shift operators. Based on the visible form of the `.c` files.

Table 1182.1: Occurrence of shift operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	>>	**_int**	29.4	**unsigned char**	<<	**_int**	2.8
_int	<<	**_int**	27.1	**_long**	<<	**_long**	2.8
unsigned int	>>	**_int**	26.1	**unsigned int**	>>	**int**	2.6
_long	<<	**_int**	11.9	**_int**	>>	**_int**	2.5
int	<<	**_int**	11.8	**int**	>>	**int**	2.1
unsigned long	>>	**_int**	11.3	**long**	>>	**_int**	2.0
_int	<<	**int**	7.3	**unsigned long**	>>	**int**	1.8
unsigned short	>>	**_int**	7.0	**unsigned long**	<<	**_int**	1.8
other-types	>>	other-types	6.9	**long**	>>	**int**	1.7
int	<<	**int**	6.0	**_unsigned long**	<<	**int**	1.3
other-types	<<	other-types	5.8	**unsigned int**	>>	**unsigned int**	1.2
unsigned int	<<	**int**	5.3	**signed long**	>>	**_int**	1.2
_unsigned long	<<	**_int**	4.9	**unsigned short**	<<	**_int**	1.1
unsigned int	<<	**_int**	4.2	**long**	<<	**_int**	1.1
unsigned char	>>	**_int**	4.0	**int**	<<	**unsigned long**	1.1
unsigned long	<<	**int**	3.8				

Semantics

1183 The integer promotions are performed on each of the operands.

shift operator integer promotions

Commentary

This is the only implicit conversion performed on the operands.

Common Implementations

The shift instructions on some processors can operate on operands of various widths, provided both widths are the same (e.g., those on the Intel x86 processor family can operate on either 8, 16, or 32 bit operands). However, the result returned by these instructions usually has the same width as the operand. For instance, shifting an operand having type **unsigned char** and value 0xf0 left by two bits returns 0xc0 rather than 0x3c0. If the next operation is an assignment to an object having the same type as the operand type, an optimizer might choose to make use of one of these narrower-width instructions; otherwise, the width corresponding to the promoted type has to be used. The width of the left operand will be the same as the object assigned to when the compound assignment form of these operators is used.

1288 assignment-expression syntax

Coding Guidelines

The type of the left operand needs to be taken into account when thinking about the result of a shift operation. However, some developers have a mental model that does not include the integer promotions. The effect of the integer promotions on an object having type **`signed char`** or **`short`** (when `sizeof(short)!= sizeof(int)`), and a negative value, is to increase the number nonzero bits in the value representation. The issue of right-shifting negative values is discussed elsewhere. Using shift operators to perform arithmetic operations is making use of representation information; this is covered by a guideline recommendation. In this case unsigned types need to be used and a deviation is provided to cover this situation.

right-shift 1196
negative value representation 569.1
representation information using object 480.1
int type only

The type of the result is that of the promoted left operand. 1184

Commentary

The usual arithmetic conversions are not performed on the operands. The right operand does not affect the type of the left operand.

Common Implementations

In some prestandard implementations the usual arithmetic conversions were performed on both operands, which meant the result type might not have been that of the promoted left operand.

Example

```
1  extern unsigned long glob;
2
3  void f(void)
4  {
5  unsigned int loc;
6
7  /*
8   * The result of the << operation is int (bits may be shifted off the
9   * end, which might not occur if loc were promoted to unsigned long),
10  * which is then promoted to type unsigned long.
11  */
12 loc = glob % (loc << glob);
13 }
```

If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined. 1185

Commentary

This specification takes into account the variation in behavior of processor shift instructions when asked to shift by a negative amount. For instance, some processors require the shift amount to be loaded into a special register, which is only capable of holding a limited range of values. Loading a value greater than the width of the operand into one of these registers may result in a shift amount of zero, while loading a negative value may result in the largest shift amount.

Other Languages

Java ensures that the value of the right operand is always in range by specifying that the equivalent of a bitwise-AND is performed on it (the number of bits extracted depending on the type of the left operand). Algol is unusual in that it defines shifting by a negative amount as reversing the direction of the shift operator (e.g., `x << -2` shifts x right by 2).

Common Implementations

The behavior may be decided by the translator or by the characteristics of the processor that executes the machine code generated to perform the operation. For instance, many translators will fold `(1 << 32)` to 0,

while generating machine code to evaluate `(y << 32)`. The Intel Pentium[627] `SAL` instruction (generated by both `gcc` and Microsoft C++ to evaluate left-shifts) only uses the bottom five bits of the shift amount (leading to `y` being shifted by zero bits in the example below).

```
1  #include <stdio.h>
2
3  int y = 1;
4
5  int main(void)
6  {
7  if ((1 << 32) != (y << 32))
8     printf("1 != %d\n", y);
9  }
```

Coding Guidelines

Developers sometimes incorrectly assume that a very large shift value will generate a result of all bits zero or all bits one (i.e., when right-shifting negative values is implemented arithmetically). Shifting by a negative amount has no common meaning associated with it (such usage is invariably a fault and nothing is served by having a guideline recommending against faults).

0 guidelines not faults

1186 89) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type.

footnote 89

Commentary

This describes common implementation practice.

1187 For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

Commentary

This describes common implementation practice.

1188 When viewed in this way, an implementation need only provide one extra byte (which may overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

Commentary

A *one past the last element* value needs to point at something. Because an implementation is not required to support the dereferencing of any pointers to this location, it need only consist of a single byte.

1172 one past the end accessing

Common Implementations

Most implementations do not allocate storage for this *one past the last element* location. In many cases such an address is the first byte of the next object defined in the source. Using the address of another object as the value of the *one past the last element* location makes it difficult for execution-time verification of pointer usage. Such a checking implementation would either have to leave some extra storage at the end of every object or use an alternative representation. The Model Implementation C Checker[681] uses a special region of storage to represent this concept. It contains information on which object is being pointed one past of, and the execution-time system knows that all pointers to this area of storage are pointing one past the end of some object.

1189 The result of **`E1 << E2`** is **`E1`** left-shifted **`E2`** bit positions;

Commentary

A left-shift moves bits from less significant positions to more significant positions within the representation. Once bits are shifted passed the most significant bit, they cease to be part of the value representation. Unlike the arithmetic operators, it is accepted that bits may be *lost* during a shift operation.

Common Implementations

Many processors contain instructions for shifting by a constant amount (the value is embedded in the instruction). Others require that the amount to shift by be loaded into a register. Some offer both forms (e.g., Intel x86). Some processors implement this instruction such that it performs the complete shift in one cycle (a barrel shifter). Other processors may take a cycle-per-bit position shifted.

Coding Guidelines

representation information using 569.1

Left-shifting is commonly known to be equivalent, in a binary representation, to multiplying a positive value by a power of two. The issue of replacing arithmetic operations by bitwise operations is covered by the guideline recommendation dealing with the use of representation information.

---

vacated bits are filled with zeros. 1190

Commentary

The vacated bits occur in the least significant bit, as the operand is shifted left.

---

left-shift of positive value

If **E1** has an unsigned type, the value of the result is $\mathbf{E1} \times 2^{\mathbf{E2}}$, reduced modulo one more than the maximum value representable in the result type. 1191

Commentary

value representation 595

This is one of the consequences of requiring that all bits in the value representation participate in forming a value. There are no more bits visible to a shift operator shift than are visible from an arithmetic operator.

Common Implementations

This C sentence is a mathematical description of the behavior. In practice processor implementations have circuitry that shifts bits rather than multiplying them.

If **E1** has a signed type and nonnegative value, and $\mathbf{E1} \times 2^{\mathbf{E2}}$ is representable in the result type, then that is the resulting value; 1192

Commentary

positive signed integer type 495
subrange of equivalent unsigned type

A positive value, having a signed type, has the same representation as the corresponding unsigned type.

C90

This specification of behavior is new in C99; however, it is the behavior that all known C90 implementations exhibit.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

Other Languages

Java defines this to be the behavior of the left-shift operator for all values (including negative ones) of E1.

---

left-shift undefined

otherwise, the behavior is undefined. 1193

Commentary

sign bit 610
representation

A negative value incorporates a sign bit. There is no consistent behavior across all processors and, given the desire to efficiently implement the C Standard on a wide range of processors, the Committee was not able to agree on a behavior.

C90

This undefined behavior was not explicitly specified in the C90 Standard.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

Common Implementations

In practice the defined behaviors are invariably one of the following:

- sign bit is treated just like the other bits (e.g., it is shifted).
- sign bit is ignored and it remains unchanged by the shift instruction (e.g., the Unisys A Series[1390]).
- sign bit is *sticky* (i.e., as soon as a 1 is shifted through it, its value stays set at 1).

Coding Guidelines

These guidelines recommend the use of a single integer type, which is signed. This undefined behavior, for signed types, means that developers wanting to portably manipulate a scalar type as a sequence of bits are going to have to declare an object to have an unsigned type.

480.1 object int type only

Dev 480.1 An object whose value is always treated as a sequence of bits, rather than an arithmetic value, may be declared to have an unsigned type.

1194 The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions.

right-shift result

Commentary

A right-shift moves bits from most significant positions to less significant positions within the representation. Once bits are shifted passed the least significant bit, they cease to be part of the value representation. Unlike the arithmetic operators, it is accepted that bits may be *lost* during a shift operation. On processors that use either one's complement or sign and magnitude representation for the integer types, the right-shift operator is always equivalent to a divide by the appropriate power of 2. Steele[1286] gives examples of how often right-shift and division by powers of 2 are incorrectly treated as being equivalent when integer types are represented using two's complement.

Coding Guidelines

Shifting right is commonly known to be equivalent, in a binary representation, to dividing a positive value by powers of 2. The issue of replacing arithmetic operations by bitwise operations is covered by the guideline recommendation dealing with the use of representation information.

569.1 representation information using

1195 If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $E1 / 2^{E2}$.

Commentary

The issues are the same as for left-shift.

1191 left-shift of positive value

Common Implementations

This C sentence is a mathematical description of the behavior. In practice processor implementations have circuitry that shifts bits rather than dividing them.

1196 If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

right-shift negative value

Commentary

Two different forms of right-shift instruction are invariably implemented by processors. One form treats the sign bit like the other value bits and vacated bits are filled with zeros (sometimes known as a logical right-shift). The other form fills vacated bits with the value of the sign bit (sometimes known as an arithmetic right-shift). Recognizing that it is possible for implementations to predict the behavior in this case, the C Committee specified the behavior as implementation-defined.

Other Languages

Java defines the `>>` operator to use sign extension and the `>>>` operator to use zero extension.

Common Implementations

Many processors have instructions that perform either form of right-shift. In such cases the decision on which to generate is made by the translator. Vendors often provide an option to select between arithmetic and logical right-shift.

Coding Guidelines

object int type only 480.1

This issue is addressed by a deviation.

6.5.8 Relational operators

relational operators syntax


```
relational-expression:
                shift-expression
                relational-expression <  shift-expression
                relational-expression >  shift-expression
                relational-expression <= shift-expression
                relational-expression >= shift-expression
```

Commentary

The term *comparison operators* is commonly used by developers to refer to the relational operators. These operators are often combined to specify intervals. (C does not support SQL's **`between`** operator, or the Cobol form `(x > 0 and < 10)`, which is equivalent to `(x > 0 and x < 10)`.)

Other Languages

Nearly every other computer language uses these tokens for these operators. Fortran uses the tokens **`.LT.`**, **`.GT.`**, **`.LE.`**, and **`.GE.`** to represent the above operators. Cobol supports these operators as well as the equivalent keywords **`LESS THAN`**, **`LESS THAN OR EQUAL`**, **`GREATER THAN`**, and **`GREATER THAN OR EQUAL`**.

Some languages (e.g., Ada and Fortran) specify that the relational and equality operators have the same precedence level.

Common Implementations

relational operators unordered

Some implementations support the unordered relational operators **`!<`**, **`!<=`**, **`!>=`**, and **`!>`**. The NCEG also included **`!<>`** for unordered or equal; **`!<>=`** for unordered; **`<>=`** for less, equal, or greater; and defines **`!=`** to mean unordered, less or greater; which enables the 26 distinct comparison predicates defined by IEC 60559 to be used. These operators were included in the Technical Report produced by the NCEG FP/IEEE Subcommittee. The expressions `(a !op b)` and `!(a op b)` have the same logical value. Without any language or library support, `a !> b` could be implemented as: `(a != a) || (b != b) || (a <= b)`.

NCEG 0

Coding Guidelines

distance effect numeric difference

A study by Moyer and Landauer[975] found that the time taken for subjects to decide which of two single-digit values was the largest was inversely proportional to the numeric difference in their values (known as a *distance effect*).

How do people compare multi-digit integer constants? For instance, do they compare them digit by digit (i.e., a serial comparison), or do they form two complete values before comparing their magnitudes (the

so-called *holistic* model)? The following two studies show that the answer depends on how the comparisons are made:

- A study by Dehaene, Dupoux, and Mehler[340] told subjects that numbers distributed around the value 55 would appear on a screen and asked them to indicate whether the number that appeared was larger or smaller than 55 (other numbers— e.g., 65— were also used as the center point). The time taken for subjects to respond was measured. The results were generally consistent with subjects using the holistic model (exceptions occurred for values ending in zero, where subjects may have noticed that a single-digit comparison was sufficient, and when the value being compared against contained two identical digits). The response times also showed a distance effect.
- A study by Zhang and Wang[1508] told subjects that they would see two numbers, both containing two digits, on a screen and asked them to indicate whether the number that appeared on the left or the right was the largest. The time taken for subjects to respond was measured. When the largest value was less than 65, the results were generally consistent with subjects using a modified serial comparison of the digits (modified to take account of Stroop-like interference between the unit and ten's digit). When the largest value was greater than 65, a serial comparison gave a slightly better fit to the results than the modified serial comparison model.

1641 stroop effect

Other studies have found that people do not treat all relational comparisons in the same way. A so-called *symbolic distance effect* exists. This is a general effect that occurs when people compare numbers or other symbols having some measure that varies along some continuum.

symbolic distance effect

For instance, a study by Moyer and Bayer[974] gave four made-up names to four circles of different diameters. One set of four circles (the small range set) had diameters 11, 13, 15, and 17 mm, while a second set (the large range set) had diameters 11, 15, 19, and 23 mm. On the first day one group of subjects learned the association between the four made-up names and their associated circle in the small range set, while another group of subjects learned the word associations for the large range set. On the second day subjects were tested. They were shown pairs of the made-up names and had to indicate which name represented the larger circle. Their response times and error rates were measured. In both cases, response rate was faster, and error rate lower, when comparing circles whose diameters differed by larger amounts. However, performance was better at all diameter differences for subjects who had memorized an association to the circles in the large range set; they responded more quickly and accurately than subjects using the small range set. The results showed a distance effect that was inversely proportional to the difference of the area of the circles being compared.

A symbolic distance effect (which is inversely proportional to the *distance* between the quantities being compared) has also been found for comparisons involving a variety of objects that differ in some way.

Freksa[453] adapted Allen's temporal algebra[15] to take account of physical constraints on perception, enabling *cognitively plausible* inferences on intervals to be performed.[748]

Table 1197.1: Common token pairs involving relational operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier <	0.7	87.9	>= *character-constant*	3.6	1.5
identifier >=	0.2	85.9	< *integer-constant*	40.0	1.3
identifier >	0.3	85.0	> *integer-constant*	53.2	0.9
identifier <=	0.1	84.8	>= *integer-constant*	41.2	0.4
) <=	0.1	10.4	< identifier	53.9	0.4
) >=	0.1	10.1	<= *integer-constant*	41.0	0.2
) <	0.3	9.9	> identifier	40.1	0.2
) >	0.1	9.6	>= identifier	50.0	0.1
<= *character-constant*	7.1	1.7	<= identifier	45.7	0.1

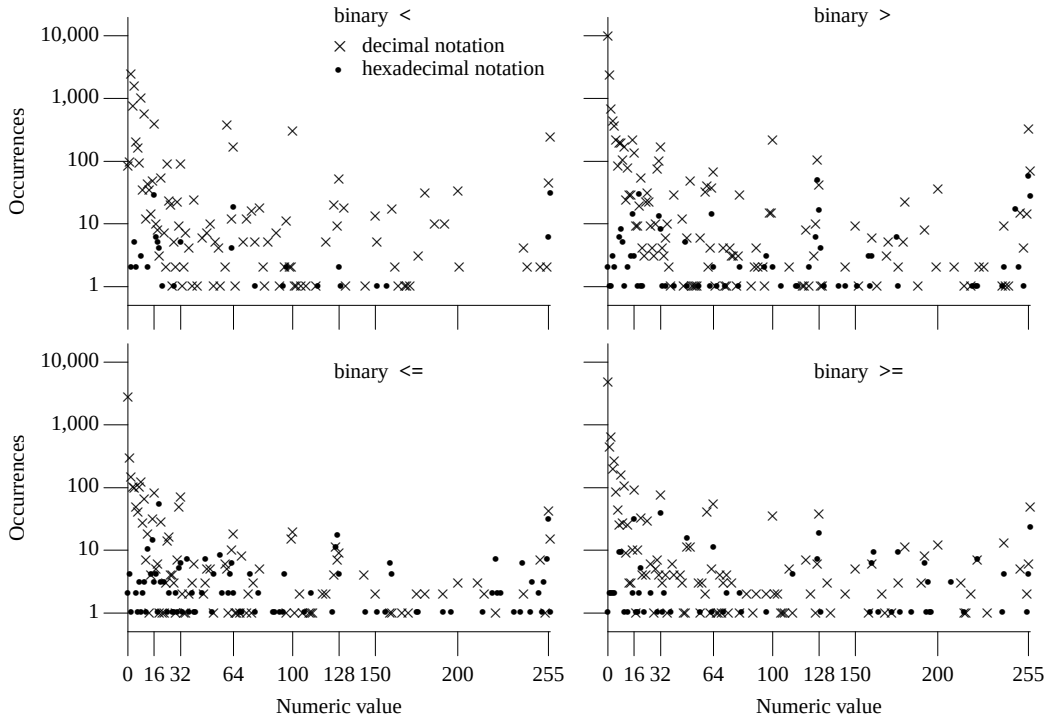


Figure 1197.1: Number of *integer-constant*s having a given value appearing as the right operand of relational operators. Based on the visible form of the `.c` files.

Table 1197.2: Occurrences (per million words) of English words used in natural language sentences expressing some relative state of affairs. Based on data from the British National Corpus.[823]

Word	Occurrences per Million Words	Word	Occurrences per Million Words
great	464	less	344
greater	154	lesser	18
greatest	51	least	45
greatly	33	–	–
–	–	less than	40

Table 1197.3: Occurrence of relational operators (as a percentage of all occurrences of the given operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	% of <	% of <=	% of >	% of >=
`if` control-expression	76.7 (3.4)	45.5 (6.7)	68.5 (1.8)	80.5 (6.0)
other contexts	11.5 (—)	4.8 (—)	9.5 (—)	8.4 (—)
`while` control-expression	4.8 (3.9)	4.6 (12.0)	4.8 (2.2)	7.6 (10.4)
`for` control-expression	7.1 (3.1)	45.2 (65.9)	17.2 (4.5)	3.5 (2.6)
`switch` control-expression	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

Constraints

relational operators constraints

One of the following shall hold:

Commentary

This list does not include pointer to function types.

Other Languages

Some languages support string values as operands to the relational operators. The comparison is usually made by comparing the corresponding characters in each string, using the designated operator.

1199 — both operands have real type;

relational operators
real operands

Commentary

Relational operators applied to complex types have no commonly accepted definition (unlike equality) and the standard does not support this usage.

1214 equality operators
arithmetic operands

Rationale

Some mathematical practice would be supported by defining the relational operators for complex operands so that `z1` *op* `z2` would be true if and only if both `creal(z1)` *op* `creal(z2)` and `cimag(z1) == cimag(z2)`. Believing such use to be uncommon, the C99 Committee voted against including this specification.

C90

both operands have arithmetic type;

The change in terminology in C99 was necessitated by the introduction of complex types.

Coding Guidelines

As discussed elsewhere, it is sometimes necessary to step through the members of an enumeration type. This suggests the use of looping constructs, which in turn implies using a member of the enumerated type in the loop termination condition.

1081 postfix operator
operand

Dev 569.1

Both operands of a relational operator may have an enumeration type or be an enumeration constant, provided it is the same enumerated type or a member of the same enumerated type.

Example

```
1  enum color {first_color, red=first_color, orange, yellow, green, blue,
2              indigo, violet, last_color};
3
4  void f(void)
5  {
6  for (enum color index=first_color; index < last_color; index++)
7     ;
8  }
```

Table 1199.1: Occurrence of relational operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	>=	**_int**	35.3	**unsigned char**	>	**_int**	2.3
int	>	**_int**	35.2	**unsigned char**	>=	**_int**	2.3
int	<	**_int**	34.8	ptr-to	<=	ptr-to	2.3
int	<=	**_int**	28.2	**unsigned int**	>=	**unsigned int**	2.1
int	<	**int**	25.5	**long**	<=	**long**	2.1
int	<=	**int**	17.5	**long**	>=	**_int**	2.0
other-types	>	other-types	15.8	**float**	>	**_int**	2.0
other-types	<	other-types	15.4	**unsigned long**	>	**unsigned long**	1.9
int	>	**int**	15.0	**unsigned short**	>	**unsigned short**	1.8
other-types	<=	other-types	14.5	**unsigned short**	>	**_int**	1.8
other-types	>=	other-types	13.2	**unsigned int**	<=	**unsigned int**	1.7
enum	<=	**_int**	12.6	ptr-to	>=	ptr-to	1.7
int	>=	**int**	10.8	**int**	<=	**unsigned long**	1.7
enum	>=	**enum**	7.5	**float**	>	**float**	1.7
unsigned int	>=	**int**	7.3	**char**	>=	**_int**	1.7
unsigned int	>	**_int**	6.0	**unsigned long**	>=	**unsigned long**	1.6
long	<	**_int**	5.3	**unsigned long**	>	**_int**	1.5
ptr-to	>	ptr-to	4.1	**double**	<=	**_double**	1.5
unsigned int	<=	**_int**	4.0	**unsigned long**	<=	**unsigned long**	1.4
unsigned int	<	**unsigned int**	3.7	**long**	>=	**long**	1.4
unsigned int	>=	**_int**	3.5	**int**	<	**unsigned long**	1.4
char	<=	**_int**	3.5	**unsigned long**	<	**unsigned long**	1.3
unsigned int	>	**unsigned int**	3.3	**long**	<	**long**	1.3
unsigned char	<=	**_int**	3.1	**_long**	>=	**_long**	1.3
long	>	**long**	2.9	**unsigned short**	<=	**unsigned short**	1.2
ptr-to	<	ptr-to	2.8	**unsigned int**	>	**int**	1.2
int	<	**unsigned int**	2.7	**float**	<	**_int**	1.2
unsigned long	<=	**_int**	2.6	**unsigned short**	<=	**_int**	1.1
unsigned int	<	**_int**	2.5	**unsigned char**	<	**_int**	1.1
_long	>=	**long**	2.5	**float**	<	**float**	1.1
long	>	**_int**	2.5	**unsigned long**	>	**int**	1.0
enum	>=	**_int**	2.5	**long**	>=	**int**	1.0
unsigned long	>=	**int**	2.4	**float**	<=	**_int**	1.0

relational operators pointer operands

— both operands are pointers to qualified or unqualified versions of compatible object types; or 1200

Commentary

subtraction 1159 pointer operands

The rationale for supporting pointers to qualified or unqualified type is the same as for pointer subtraction. Differences in the qualification of pointed-to types is guaranteed not to affect the equality status of two pointer values.

pointer 746 converting qualified/unqualified

C++

5.9p2

Pointers to objects or functions of the same type (after pointer conversions) can be compared, with a result defined as follows:

The pointer conversions (4.4) handles differences in type qualification. But the underlying basic types have to be the same in C++. C only requires that the types be compatible. When one of the pointed-to types is an enumerated type and the other pointed-to type is the compatible integer type, C permits such operands to occur in the same relational-expression; C++ does not (see pointer subtraction for an example).

subtraction 1159 pointer operands

Other Languages

Few languages support relational comparisons on objects having pointer types.

Table 1200.1: Occurrence of relational operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Op	Right Operand	%	Left Operand	Op	Right Operand	%
char *	>	**char ***	67.5	**const char ***	>	**const char ***	4.0
char *	<=	**char ***	39.6	other-types	>	other-types	3.8
char *	>=	**char ***	26.9	**int ***	>=	**int ***	3.6
char *	<	**char ***	25.8	**const char ***	>=	**const char ***	3.6
struct *	<=	**struct ***	23.2	**struct ***	>	**struct ***	3.1
unsigned char *	>=	**unsigned char ***	22.8	**short ***	<=	**short ***	3.0
unsigned char *	<	**unsigned char ***	21.0	other-types	<	other-types	2.8
short *	>=	**short ***	16.1	**unsigned int ***	>=	**unsigned int ***	2.6
struct *	<	**struct ***	14.9	**const char ***	<	**const char ***	2.6
unsigned char *	<=	**unsigned char ***	13.4	**const unsigned char ***	<	**const unsigned char ***	2.0
signed int *	<	**signed int ***	13.1	**unsigned int ***	>	**unsigned int ***	1.9
struct *	>=	**struct ***	13.0	**unsigned long ***	<=	**unsigned long ***	1.8
void *	>	**void ***	11.0	other-types	<=	other-types	1.8
void *	<	**void ***	9.4	**const char ***	<=	**const char ***	1.8
unsigned char *	>	**unsigned char ***	8.7	**void ***	>=	**void ***	1.6
unsigned short *	<=	**unsigned short ***	7.9	**unsigned short ***	<	**unsigned short ***	1.2
const unsigned char *	<=	**const unsigned char ***	4.9	**unsigned int ***	<	**unsigned int ***	1.2
ptr-to *	<	ptr-to *	4.8	**union ***	<=	**union ***	1.2
unsigned short *	>=	**unsigned short ***	4.7	**int ***	<	**int ***	1.2
const unsigned char *	>=	**const unsigned char ***	4.7	**int ***	<=	**int ***	1.2

1201 — both operands are pointers to qualified or unqualified versions of compatible incomplete types.

relational operators pointer to incomplete type

Commentary

Because the operands are pointers to compatible types, a relational operator only needs to compare the pointer values (i.e., information on the pointed-to type is not needed to perform the comparison).

C++

C++ classifies incomplete object types that can be completed as object types, so the discussion in the previous C sentence is also applicable here. 475 object types

Other Languages

Those languages that do not support pointer arithmetic invariably do not support the operands of the relational operators having pointer type.

Example

```
1  extern int a[];
2
3  void f(void *p)
4  {
5  if (a+1 > p)   /* Constraint violation. */
6     ;
7  if (a+1 > a+2) /* Does not affect the conformance status of the program. */
8     ;
9  }
```

Semantics

relational operators
usual arithmetic conversions
arithmetic 710 conversions
integer promotions

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. 1202

Commentary

This may also cause the integer promotions to be performed.

Common Implementations

The *comparison* (the term *relational* is not usually used by developers) instructions on some processors can operate on operands of various widths, provided both widths are the same (e.g., those on the Intel x86 processor family can operate on 8-, 16-, or 32-bit operands). If both operands have the same type before the usual arithmetic conversions, an implementation may choose to make use of such instructions.

Coding Guidelines

The relational operators produce some of the most unexpected results from developers' point of view. The root cause of the unexpected behavior is invariably a difference in the sign of the types after the integer promotions of the operands. Following the guideline recommendation specifying the use of a single integer type reduces the possibility of this unexpected behavior occurring. However, the use of typedef names from the standard header, or third-party libraries (or even the use of some operators), can cause of a mixture of signed types to appear as operands.

object 480.1 int type only

Cg 1202.1

The types of the two operands in a *relational-expression*, after the integer promotions are performed on each of them, shall both be either signed or both unsigned.

Example

```
#include <stdio.h>

extern int glob;

void f(void)
{
if (glob > sizeof(glob))
   {
   if (glob < (int)sizeof(glob))
      print("glob has a negative value\n");
   else
      print("glob has a positive value\n");
   }
else if (glob != 0)
   print("glob has at most 66.66....% chance of being negative\n");
}
```

relational operators pointer to object

For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. 1203

Commentary

additive 1165 operators pointer to object

The same sentence appears elsewhere in the standard and the issues are discussed there.

C++

This wording appears in 5.7p4, Additive operators, but does not appear in 5.9, Relational operators. This would seem to be an oversight on the part of the C++ committee, as existing implementations act as if the requirement was present in the C++ Standard.

1204 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to.

Commentary

Here the term *address space* refers to the object within which the two pointers point. It is not being used in the common usage sense of the address space of the program, which refers to all the storage locations available to an executing program. The standard does not define the absolute location of any object or subobject. However, in some cases it does define their locations relative to other subobjects (and the relational operators are about relative positions).

1206 structure members later compare later

C++

The C++ Standard does not make this observation.

Common Implementations

In most implementations all objects have a location relative to all other objects in the storage used by a program. Processors rarely perform any checks on the values of pointers. Pointers to different storage locations (which may or may not currently be used to hold an object) return a result based on these relative positions in that storage.

Coding Guidelines

Having provided a mechanism to index subcomponents of an object, relational operations on the values of indexing expressions is the same whether the types involved are integers or pointers.

1205 If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal.

Commentary

Relational comparison of pointer values has a meaningful interpretation in C because of the language's support for arithmetic on pointer values. A more detailed specification of pointer equality is given elsewhere in the standard.

1233 pointers compare equal

Some expressions, having pointer type, can be paired as operands of a relational operator but not as operands of the subtraction operator; for instance, given the declarations:

```
struct S {
           int    mem_1;
           double mem_2[5];
           int    mem_3;
           double mem_4[5];
         } x,
           y[10];
```

then the following pairs of expressions may appear together as operands of the relational operators, but not as operands of the subtraction operator:

```
        x   .mem_1    op   x   .mem_3
        y[1].mem_1    op   y[3].mem_3
        y[1].mem_1    op   y[3].mem_3
        x   .mem_2[1] op   x   .mem_4[1]
```

C++

This requirement can be deduced from:

5.9p2

— *If two pointers* `p` *and* `q` *of the same type point to the same object or function, or both point one past the end of the same array, or are both null, then* `p<=q` *and* `p>=q` *both yield* **`true`** *and* `p<q` *and* `p>q` *both yield* **`false`**.

structure members later compare later
array elements later compare later

If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. 1206

Commentary

member 1422 address increasing

The standard does not specify a representation for pointer types, but rather some of the properties they must have. The relative addresses of members of the same structure type are specified elsewhere. This wording specifies that increasing the value of an address causes it to compare greater than the original address, provided both addresses refer to the same structure object. This requirement is not sufficient to calculate the address of subsequent members of a structure type. For instance, the expression `&x.m1+sizeof(x.m1)` does not take into account any padding that may occur after the member `m1`. There is existing source code that uses pointer arithmetic to access the members of a structure object. However, the more widespread usage is interpreting the same object using different types. This requirement and the creation of the `offsetof` macro codifies existing practice.

There are two possible ordering of array elements in storage. Established practice, prior to the design of C, was for increasing index values to refer to elements ever further away from the first. This C sentence specifies this implementation behavior.

structure 1424 unnamed padding

Nothing is said about the result of comparing pointers to any padding bytes. Neither is anything said about the behavior of relational operators when their operands point to different objects.

C++

This requirement also applies in C++ (5.9p2). If the declaration of two pointed-to members are separated by an access-specifier label (a construct not available in C), the result of the comparison is unspecified.

Other Languages

The implementation details of array types in Java is sufficiently opaque that storage for each element could be allocated on a different processor, or in contiguous storage locations on one processor.

Coding Guidelines

pointer 1167 arithmetic addition result

Common existing practice, for C developers, is to use the less than rather than the not equal operator as a loop termination condition (see Table 1763.1). Pointer arithmetic is based on accessing the elements of an array, not its bytes. Looping through an array object using pointer arithmetic and relational operators has a fully defined behavior.

Example

```
1  struct T {
2            int first;
3            /* Some member declarations. */
4            int middle;
5            /* More member declarations. */
6           } glob;
7
8  void f(void)
9  {
10 int *p_loc = &glob.first;
11
12 /*
```

```
13    * Set all members before middle to zero.
14    */
15   while (p_loc < &glob.middle)
16      {
17      *p_loc=0;
18      p_loc++;
19      }
20   }
```

1207 All pointers to members of the same union object compare equal.

pointer to union members compare equal

Commentary

This behavior can also be deduced from pointers to the members of the same union type also pointing at the union object and pointers to the same object comparing equal.

1427 union members start same address
1233 pointers compare equal

1208 If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`.

Commentary

This special case deals with pointers that do not point at the same object (but have an association with the same object).

C90

The C90 Standard contains the additional words, after those above:

> *even though Q+1 does not point to an element of the array object.*

Common Implementations

On segmented architectures incrementing a pointer past the end of a segment causes the address to wrap around to the beginning of that segment (usually address zero). If an array is allocated within such a segment, either the implementation must ensure that there is room after the array for there to be a one past the end address, or it uses some other implementation technique to handle this case (e.g., if the segment used is part of a pointer's representation, a special *one past the end* segment value might be assigned).

590 pointer segmented architecture

1209 In all other cases, the behavior is undefined.

relational pointer comparison undefined if not same object

Commentary

The C relational operator model enables pointers to objects to be treated in the same way as indexes into array objects. Relational comparisons between indexes into two different array objects (that are not both subobjects of a larger object) rarely have any meaning and the standard does not define such support for pointers. Some applications do need to make use of information on the relative locations of different objects in storage. However, this usage was not considered to be of sufficient general utility for the Committee to specify a model defining the behavior.

C90

If the objects pointed to are not members of the same aggregate or union object, the result is undefined with the following exception.

C++

5.9p2

— *Other pointer comparisons are unspecified.*

Source developed using a C++ translator may contain pointer comparisons that would cause undefined behavior if processed by a C translator.

Common Implementations

Most implementations perform no checks prior to any operation on values having pointer type. Most processors use the same instructions for performing relational comparisons involving pointer types as they use for arithmetic types. For processors that use a segmented memory architecture, a pointer value is often represented using two components, a segment number and an offset within that segment. A consequence of this representation is that there are many benefits in allocating storage for objects such that it fits within a single segment (i.e., storage for an object does not span a segment boundary). One benefit is an optimization involving the generated machine code for some of the relational operators, which only needs to check the segment offset component. This can lead to the situation where `p >= q` is false but `p > q` is true, when `p` and `q` point to different objects.

pointer 590 segmented architecture

Coding Guidelines

Developers sometimes do intend to perform relational operations on pointers to different objects (e.g., to perform garbage collection). Such usage makes use of information on the layout of objects in storage, this issue is discussed elsewhere.

storage 1354 layout

---

relational operators result value

Each of the operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[90)] 1210

Commentary

The four comparisons (<, <=, >=, >) raise the invalid exception if either operand is a NaN (and returns 0). Some implementations support what are known as *unordered* relational operators. The floating-point comparison macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`), new in C99, can be used in those cases where NaNs may occur. They are similar to the existing relational operators, but do not raise invalid for NaN operands.

relational 1197 operators unordered

C++

5.9p1 *The operators* < *(less than),* > *(greater than),* <= *(less than or equal to), and* >= *(greater than or equal to) all yield* **`false`** *or* **`true`**.

This difference is only visible to the developer in one case, the result type. In all other situations the behavior is the same; false and true will be converted to 0 and 1 as-needed.

relational 1211 operators result type

Other Languages

Languages that support a boolean data type usually specify **`true`** and **`false`** return values for these operators.

Common Implementations

The constant 0 is commonly seen as an operand to these operators. The principles used for this case generally apply to all other combinations of operand (see the logical negation operator for details). When one of the operands is not the constant 0, a comparison has to be performed. Most processors require the two operands to be in registers. (A few processors[968] support instructions that compare the contents of storage, but the available addressing modes are usually severely limited.) The comparison of the contents of the two specified registers is reflected in the settings of the bits in the condition flags register. RISC processors do not contain instructions for comparing integer values, the subtract instruction is used to set condition flags (e.g., `x > y` becoming `x-y > 0`). Most processors contain compare instructions that include a small constant as part of their encoding. This removes the need to load a value into a register.

logical 1111 negation result is

Relational operators are often used to control the number of iterations performed by a loop. The implementation issues involved in this usage are discussed elsewhere.

iteration 1763 statement syntax

Coding Guidelines

While the result is specified in numeric terms, most occurrences of this operator are as the top-level operator in a controlling expression (see Table 1197.3). These contexts are usually treated as involving a boolean role, 476 boolean role rather than a numeric value.

1211 The result has type **int**.

relational operators result type

Commentary

The first C Standard did not include a boolean data type. C99 maintains compatibility with this existing definition.

C++

The type of the result is ***bool****.* 5.9p1

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

Other Languages

Languages that support a boolean type usually specify a boolean result type for these operators.

Coding Guidelines

Most occurrences of these operators are in controlling expressions (see Table 1197.3), and developers are likely to use a thought process based on this common usage. Given this common usage developers often don't need to consider the result in terms of delivering a value having a type, but as a value that determines the flow of control. In such a context the type of the result is irrelevant. Some of the issues that occur when the result of a relational operation is an operand of another operator (e.g., `x=(a<b)`) are discussed elsewhere. 476 boolean role

6.5.9 Equality operators

---

equality operators syntax

1212

```
equality-expression:
                relational-expression
                equality-expression == relational-expression
                equality-expression != relational-expression
```

Commentary

The use of the term *equality* is based on the most commonly occurring of the two operators. The terms *equals* and *not equals* are commonly used by developers.

Other Languages

Those languages that use the character sequence **:=** to represent assignment usually use the **=** character to represent the equality operator token. Some languages use the character sequence **<>** to represent the not equal operator. Fortran uses the tokens **.EQ.** and **.NE.** to represent the quality operators.

Pascal supports the **IN** operator. Instead of comparing one value against another, this operator provides a mechanism for testing whether a value is **IN** a set of values. For instance, `i IN [4, 6]` tests whether `i` has the value 4 or 6; `i IN [2..5]` tests whether `i` has a value between 2 and 5, inclusive.

Some languages offer a number of different ways of comparing for equality. For instance, in Scheme `(eq? X Y)` tests whether `X` and `Y` refer to the same object, but `(equal? X Y)` tests, recursively, whether `X` and `Y` have the same structure (which for linked data structures involves walking them, comparing each pair of nodes).

Perl supports the **<=>** operator, which returns -1 if the left operand is less than the right operand, 0 if they are equal, and 1 if the left operand is greater than the right.

Coding Guidelines

A study by Dehaene and Akhavein[339] asked subjects to make same/difference judgments for pairs of numeric values. The values were either presented in Arabic form (e.g., 2 2), as written words (e.g., TWO TWO), or in mixed form (e.g., 2 TWO, or TWO 2). The two numeric values were either the same, close to each other (e.g., 1 2), or not close to each other (e.g., THREE NINE). The time taken for subjects to make a same/different judgment was measured, along with their error rate.

distance effect 1197
numeric difference

The results (see Figure 1212.1) for different the kinds of numeric value pairings followed the same pattern. This pattern is consistent with both the Arabic and written-word representation being converted to a common, internal, magnitude representation (in the subject's head). This *distance effect* is discussed elsewhere.

Numeric values appear in source code in written form through the use of macro definitions. However, the common usage is to use a symbolic name to represent some numeric quantity, not the spoken words of the value; for instance, using the name `MAX_LINE_LENGTH` to denote the maximum number of characters that can appear on a line. Source code may contain a comparison of such a name against an integer constant (e.g., `#if MAX_LINE_LENGTH == 80`) or against another symbolic name (e.g., `#if MAX_LINE_LENGTH == XYZ_TERM_LINE_LENGTH`). Comparison of two decimal constants appearing in the visible source as Arabic numerals is rare. The affect of reader's prior knowledge of the current value of `MAX_LINE_LENGTH` on their processing of this equality operation is unknown. The Dehaene and Akhavein study showed that using symbolic names for numeric values does not prevent the distance effect from occurring.

An equality comparison may involve performing a mental calculation on one of the operands. Within source code, this may be a *what if?* kind of calculation or the values of the operands may be known at translation time and a reader may be attempting to verify if a condition is true or false (e.g., which arm of a conditional inclusion directive will be processed by a translator).

A study by Zbrodoff and Logan[1503] measured subjects' performance in arithmetic verification tasks (e.g., $3\times5 = 17$, true/false?). The intent was to test the hypothesis that such verification tasks consisted of two stages: first performing the arithmetic operation and then comparing the computed value against the value given. In most cases the results were not consistent with this two-stage production/comparison process, but were consistent with verification involving a comparison of all the information presented (i.e., including the putative answer) against memory.

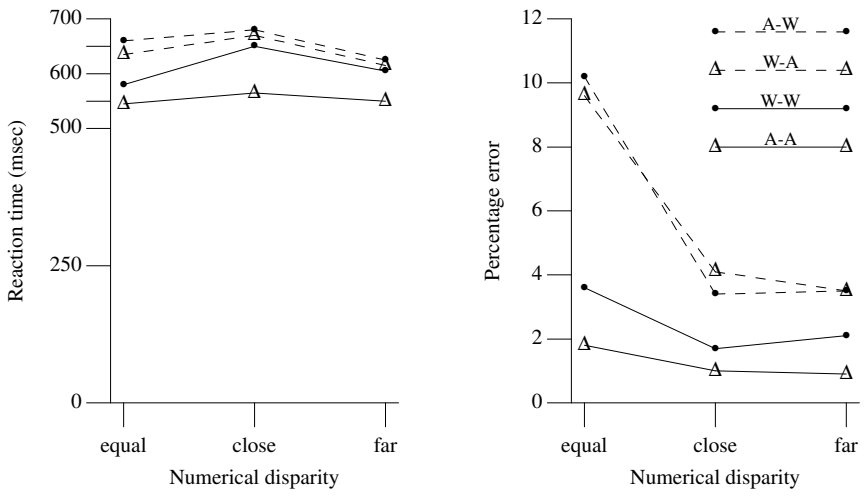


Figure 1212.1: Reaction time (in milliseconds) and error rates for same/different judgment for values between one and nine, expressed in *A*rabic or *W*ord form. Adapted from Dehaene and Akhavein.[339]

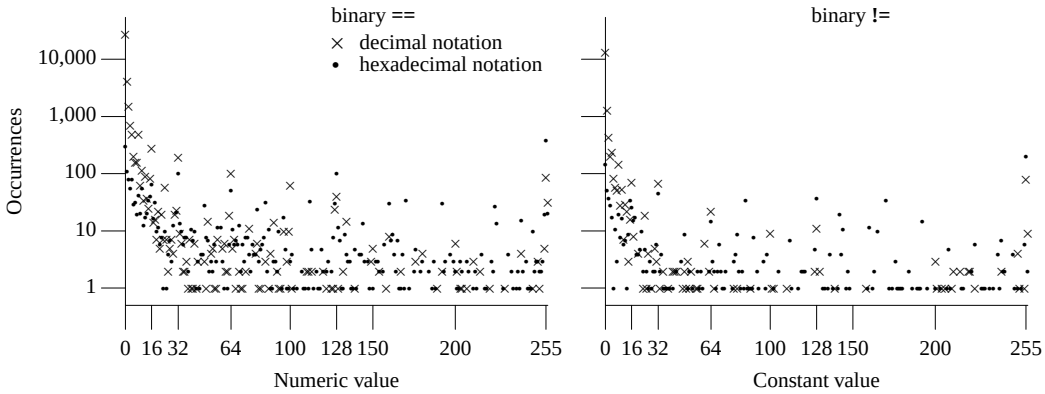


Figure 1212.2: Number of *integer-constant*s having a given value appearing as the right operand of equality operators. Based on the visible form of the `.c` files.

Table 1212.1: Common token pairs involving the equality operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have a very low percentage are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier `!=`	0.6	69.2	`]` `!=`	1.4	5.1
identifier `==`	1.2	67.9	`==` `-v`	2.6	3.5
`)` `==`	1.6	25.1	`==` *integer-constant*	25.5	2.0
`)` `!=`	0.8	24.7	`==` identifier	62.1	1.1
`==` *character-constant*	7.1	22.8	`!=` *integer-constant*	22.7	0.9
`!=` *character-constant*	5.3	8.4	`!=` identifier	65.0	0.6
`]` `==`	3.1	5.6			

Constraints

1213 One of the following shall hold:

equality operators constraints

Commentary

This list is very similar to that given for simple assignment, except that there is no support for equality operations on structure or union types.

1296 simple assignment constraints
1298 assignment structure types

The C89 Committee considered, on more than one occasion, permitting comparison of structures for equality. Such proposals foundered on the problem of holes in structures. A byte-wise comparison of two structures would require that the holes assuredly be set to zero so that all holes would compare equal, a difficult task for automatic or dynamically allocated variables. The possibility of union-type elements in a structure raises insuperable problems with this approach. Without the assurance that all holes were set to zero, the implementation would have to be prepared to break a structure comparison into an arbitrary number of member comparisons; a seemingly simple expression could thus expand into a substantial stretch of code, which is contrary to the spirit of C.

Rationale

C++

The `==` *(equal to) and the* `!=` *(not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators . . .*

5.10p1

See relational operators.

1198 relational operators constraints

Other Languages

Other languages include support for equality operations on structures, strings, sets, and even linked lists.

---

equality operators arithmetic operands — both operands have arithmetic type; 1214

Commentary

These operators are defined for operands having a complex type.

Coding Guidelines

The equality operators are different from the relational operators in that an exact match is being tested for. In the case of the real and complex types, testing for an exact match does not always make sense. Rounding and other types of error in operations on floating-point values means that an exact value can never be expected, only a very close approximation.

The equality operator is sometimes used to check a property of floating-point numbers. Is one value so small, compared to a second value, that the effect of adding them together is to deliver the larger one as the result? In other words is the smaller value insignificant, compared to the larger value? This test can be made because we are interested in the smaller value, not the larger one. The natural logarithm of one plus a very small value is equal, within a reasonable error bound, to that very small value.[500]

```
1  #include <math.h>
2
3  double log_x_plus_1(double x)
4  {
5  if (1.0 + x == 1.0)
6     return x;
7  else
8     return log(1.0 + x) * x / ((1.0 + x) - 1.0);
9  }
```

The above algorithm relies on the value of the expression `1.0+x` calculated in the equality test being identical to the value passed as an argument to the log function. The value passed to the `log` function has type **`double`**. If the equality test is carried out using a precision that is different from **`double`** (for instance, using an extended precision), it is possible that the extra bits available in the result will cause the equality test to fail, invoking the `log` function which returns zero (the *log* of 1.0, since rounding to **`double`** will remove any extended precision bits)— a value that has a relative error of one.

FLT_EVAL_METHOD 354

This example shows that using equality operators on values having real type requires more than a numerical analysis of the algorithm being used. (Goldberg[500] provides a readable analysis of error bounds.) What is also needed is knowledge of how an implementation actually performs the calculation. In this case use of extended precision can invalidate all of the error bounds deduced by careful numerical analysis.

Cg 1214.1 The operands of the equality operators shall not have floating-point or complex type.

Dev 1214.1 A numerical algorithm that performs equality tests on floating-point values may be used if an implementation always evaluates lexically identical expressions to the same result.

Comparing values having an enumerated type for equality (or inequality) need not imply any use of representation information (e.g., relative ordering of members).

Dev 569.1 Both operands of an equality operator may have an enumeration type or be an enumeration constant, provided it is the same enumerated type or a member of the same enumerated type.

Table 1214.1: Occurrence of equality operators having particular operand types (as a percentage of all occurrences of each operator). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
ptr-to	!=	ptr-to	28.5	**char**	!=	**_int**	3.9
int	==	**_int**	21.1	ptr-to	!=	**_int**	3.5
int	!=	**_int**	15.8	**unsigned long**	!=	**unsigned long**	2.5
ptr-to	==	ptr-to	15.3	**unsigned long**	!=	**_int**	2.2
other-types	==	other-types	12.7	**unsigned short**	!=	**_int**	2.0
other-types	!=	other-types	12.6	int:16 16	!=	**_int**	2.0
unsigned char	==	**_int**	9.5	**unsigned short**	!=	**unsigned short**	1.9
enum	==	**_int**	9.1	**unsigned int**	!=	**unsigned int**	1.9
int:16 16	==	**_int**	8.2	ptr-to	==	**_int**	1.8
int	!=	**int**	6.5	**unsigned short**	==	**_int**	1.7
int	==	**int**	6.5	**unsigned long**	==	**unsigned long**	1.7
char	==	**_int**	5.5	**unsigned long**	==	**_int**	1.6
unsigned char	!=	**_int**	4.8	**unsigned long**	!=	**_long**	1.3
enum	!=	**_int**	4.8	**unsigned char**	!=	**unsigned char**	1.3
unsigned int	!=	**_int**	4.4	**unsigned int**	==	**unsigned int**	1.1
unsigned int	==	**_int**	4.0				

1215 — both operands are pointers to qualified or unqualified versions of compatible types;

equality operators pointer to compatible types

Commentary

The rationale for supporting pointers to qualified or unqualified type is the same as for pointer subtraction and relational operators. Differences in the qualification of pointed-to types is guaranteed not to affect the equality status of two pointer values.

1159 subtraction pointer operands
1200 relational operators pointer operands
746 pointer converting qualified/unqualified

The operands may have a pointer to function type.

C++

1200 relational operators pointer operands

The discussion on the relational operators is applicable here.

Other Languages

Languages containing a pointer data type allow values having such types to be compared for equality and inequality.

Common Implementations

Because the operands need not refer to the same objects, it is necessary to compare all the information represented in a pointer value (e.g., segment and offset in a segmented architecture). The constraint that both pointers point to compatible types ensures that, in those cases where different representations are used for pointers to different types, an implementation does not have to concern itself with implicitly converting one representation to another.

590 pointer segmented architecture

In implementations that use a flat address space an equality operator is usually implemented using one of the unsigned integer comparison machine instructions.

Table 1215.1: Occurrence of equality operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type; an _ prefix indicates a literal operand, *_int* is probably the 0 representation of the null-pointer constant). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
struct *	==	_ **int**	59.9	**int** *	!=	_ **int**	3.0
struct *	!=	_ **int**	52.2	**void** *	==	_ **int**	2.2
union *	!=	_ **int**	18.3	**const char** *	==	_ **int**	1.8
union *	==	_ **int**	18.1	**int**	==	**void** *	1.4
other-types	==	other-types	8.1	**const char** *	!=	_ **int**	1.4
char *	!=	_ **int**	8.1	**int**	!=	**void** *	1.3
char *	==	_ **int**	7.3	**unsigned char** *	==	_ **int**	1.1
array-index	!=	**void** *	6.9	ptr-to *	!=	_ **int**	1.1
other-types	!=	other-types	6.4	**char** *	!=	array-index	1.1

equality operators pointer to incomplete type

— one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**; or 1216

Commentary

This combination of operands supports the idea that pointer to **void** represents a generic container for any pointer type. Relaxing the constraint in the previous C sentence, requiring the pointer types to be compatible, removes the need for an explicit cast of the operand having pointer to **void** type. A pointer to **void** is only guaranteed to compare equal to the original pointer value when it is converted back to that value's original pointer type.

converted 745 via pointer to void compare equal

C++

This special case is not called out in the C++ Standard.

```
#include <stdlib.h>

struct node {
            int mem;
            };
void *glob;

void f(void)
{
/* The following is conforming */
// The following is ill-formed
struct node *p = malloc(sizeof(struct node));

/*
 * There are no C/C++ differences when the object being assigned
 * has a pointer to void type, 4.10p2.
 */
glob = p;
}
```

relational 1198 operators constraints

See relational operators for additional issues.

Other Languages

Some languages include the concept of a generic pointer type, with some operators handling operands having this type.

Coding Guidelines

The use of pointer to **void** as a generic pointer type is common developer knowledge. There is no benefit in casting the operand having this type to the type of the other operand. An explicit cast creates a possible future

maintenance cost, unless the pointer type is denoted by a typedef name. (In this case a single change to the definition of the typedef name changes all uses; otherwise, a change of pointer type requires all corresponding occurrences in the source to be changed.)

1217 — one operand is a pointer and the other is a null pointer constant.

equality operators null pointer constant

Commentary

The previous C sentence excluded the use of operands having pointer to function type. An equality test against the null pointer constant is a common loop termination condition when walking a dynamic data structure. This permission allows the loop termination test to include an operand having a pointer to function type. (It also covers the case of 0 being used to denote the null pointer constant.)

748 null pointer constant

Other Languages

Languages that support pointer data types invariably have some form of null pointer that can be compared for equality against all other pointer types.

Semantics

1218 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.[91)]

Commentary

They differ from the relational operators in their handling of NaN. The equality operators are defined not to raise an exception if one or more operators is NaNs, while the relational operators do raise an exception.

1210 relational operators result value

Common Implementations

The processor instructions also perform in an analogous way to the relational operator instructions and translators generate analogous instruction sequences.

Coding Guidelines

Equality testing is one of the most fundamental operations people perform. In many contexts it is possible to assign a generally accepted meaning to an equality comparison, but not to a relational comparison. The C equality operators support a small subset of the possible equality tests (i.e., those between values having arithmetic and pointer types). Within this subset, it is analogous to the relational operators. (However, these C relational operations differ from the equality operators in that they often make up a significant percentage of the relational tests performed in a program; the other tests usually are between strings, often performed by calls to library functions.)

In most cases developers have to write code to perform equality tests when the operands do not have arithmetic types (e.g., to test whether two lists, or two arrays, are equal). However, there is one nonarithmetic type that is sometimes represented in an object having an integer type— a set. Representing a set type using an integer type creates the possibility for the equality operators to be inappropriately used. The following discussion looks at some of the difficulties of working out whether the use of an equality operator was intended when the operands appear to be sets.

The enumeration constants defined in an enumerated type definition may be assigned values intended to fill a variety of roles. For instance, their representation may be numerically distinct because objects of that type are only intended to represent a single member, or the representation may be bitwise distinct, because objects of that type are intended to be able to represent more than one member at the same time. (A set type is created by ORing together different members.) In the latter case the operations performed by the C equality operators do not correspond to the most commonly seen tests, that of set membership (based on experience with Pascal, which supports a set type).

517 enumeration set of named constants

```
1  #define in_set(x, y)  (((x) & (y)) == (x))
2
3  enum T {mem_1 = 0x01, mem_2 = 0x02, mem_3 = 0x04, mem_4 = 0x10};
```

```
4
5  extern enum T glob;
6
7  void f(enum T p_1)
8  {
9  if (in_set(mem_2, p_1)) /* does p_1 include the member mem_2? */
10    ; /* ... */
11 /*
12  * The following test can be interpreted as either:
13  *    1) are the two sets equal?
14  *    2) does p_1 only include the member mem_2?
15  */
16 if (p_1 == mem_2)
17    ; /* ... */
18 }
```

How set types are represented as C types, and the interpretation given to C operators having operands of these types, is considered to be a level of abstraction that is outside the scope of these coding guideline subsections.

---

equality operators true or false

Each of the operators yields 1 if the specified relation is true and 0 if it is false. 1219

Commentary

`x != x` yields 0, except when `x` is a NaN (when it yields 1). `x == x` yields 1, except when `x` is a NaN (when it yields 0). Possible limits on how strongly equality can be interpreted are discussed elsewhere.

footnote 602 43 subtraction 1164 result of

C++

5.10p1 *The `==` (equal to) and the `!=` (not equal to) operators have the same . . . truth-value result as the relational operators.*

equality 1220 operators result type

This difference is only visible to the developer in one case. In all other situations the behavior is the same—false and true will be converted to 0 and 1 as needed.

Other Languages

Languages that support a boolean data type usually specify **`true`** and **`false`** return values for these operators.

Example

```
1  #include <math.h>
2
3  _Bool f(float x, float y)
4  {
5  if (isnan(x) && isnan(y))
6     return 1;
7  return x==y;
8  }
```

---

equality operators result type

The result has type **`int`**. 1220

Commentary

relational 1211 operators result type

The rationale for this choice is the same as for relational operators.

C++

5.10p1

The == (equal to) and the != (not equal to) operators have the same . . . result type as the relational operators.

The difference is also the same as relational operators.

1211 relational operators result type

Other Languages

Languages that support a boolean type usually specify a boolean result type for these operators.

Coding Guidelines

The coding guideline issues are the same as those for the relational operators.

1211 relational operators result type

1221 For any pair of operands, exactly one of the relations is true.

equality operators exactly one relation is true

Commentary

This is a requirement on the implementation. No equivalent requirement is stated for relational operators (and for certain operand values would not hold). It is a moot point whether this requirement applies if both operands have indeterminate values since accessing either of them causes undefined behavior. It might be more accurate to say "for an evaluation of any pair of operands . . . ", since one or more of the operands may be volatile-qualified.

1210 relational operators result value

1476 type qualifier syntax

C90

This requirement was not explicitly specified in the C90 Standard. It was created, in part, by the response to DR #172.

C++

This requirement is not explicitly specified in the C++ Standard.

Example

```
#include <limits.h>

void f(void)
{
int i = INT_MIN;

if (i != INT_MIN)
   printf("This sentence will never appear\n");
if ((float)i != INT_MIN)
   printf("This sentence might appear\n");

while ((i != 0) && (i == -i)) /* When using two's compliment this is an infinite loop. */
   { /* i not modified in loop. */ }
}
```

1222 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

Commentary

This may also cause the integer promotions to be performed.

710 arithmetic conversions integer promotions

C90

Where the operands have types and values suitable for the relational operators, the semantics detailed in 6.3.8 apply.

relational operators 1202
usual arithmetic conversions

Coding Guidelines

The unexpected results that can occur for the relational operators (when the operands have differently signed types), as a consequence of these conversions, are much less likely to occur for the equality operators. For instance, `signed_object == unsigned_object` is true when both objects have the same value and when the implicit conversion of a negative value to unsigned results in the same value (e.g., `UINT_MAX == -1`).

Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. 1223

Commentary

complex 506
component representation

Given the representation used for complex types, this form of equality testing is the obvious choice (had polar form been used, the obvious choice would have been to compare their modulus and angles).

C90

Support for complex types is new in C99.

Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal. 1224

Commentary

usual arithmetic conversions 706
real type 700
converted to complex

The usual arithmetic conversions specifies the type of the result. The details of real type to complex type conversions are specified elsewhere.

C90

Support for different type domains, and complex types, is new in C99.

C++

real type 700
converted to complex

The concept of type domain is new in C99 and is not specified in the C++ Standard, which defines constructors to handle this case. The conversions performed by these constructions have the same effect as those performed in C.

equality operators 1214.1
not floating-point operands

Coding Guidelines

The guideline recommendation dealing with the comparison of floating-point values for equality is applicable here.

---

footnote 91

91) The expression **a<b<c** is not interpreted as in ordinary mathematics. 1225

Commentary

In mathematical notation this would probably be interpreted as equivalent to the C expression `(a < b) && (b < c)`.

C++

The C++ Standard does not make this observation.

Other Languages

Cobol has a **between** operator that enables this mathematical test to be performed. BCPL allows any number of relational operators in a sequence; they're ANDed together; for instance, the BCPL expression `x > y >= z == q < r` is equivalent to the C `x > y && y >= z && z == q && q < r`.

expression 943.1
shall be parenthesized

Coding Guidelines

This issue is covered by the guideline recommendation dealing with the use of parenthesis.

As the syntax indicates, it means **(a<b)<c**; 1226

Commentary

The **<** operator associates to the left.

1227 in other words, "if **a** is less than **b**, compare 1 to **c**; otherwise, compare 0 to **c**".

Commentary

As discussed elsewhere, the evaluation order of the operands is not guaranteed to be a, b, and c. 944 expression order of evaluation

1228 90) Because of the precedences, **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth-value. footnote 90

Commentary

The grouping here is `(a<b) == (c<d)`.

Coding Guidelines

This issue is covered by the guideline recommendation dealing with the use of parentheses. 943.1 expression shall be parenthesized

1229 Otherwise, at least one operand is a pointer.

Commentary

The case where only one operand is a pointer is when the other operand is the integer constant 0 (which is interpreted as a null pointer constant). 748 null pointer constant

C++

The C++ Standard does not break its discussion down into the nonpointer and pointer cases.

1230 If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. equality operators null pointer constant converted

Commentary

If different pointer types use different representations for the null pointer, this implicit conversion ensures that the equality test is performed against the appropriate representation. 750 null pointer conversion yields null pointer

C90

If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type.

In the case of the expression `(void *)0 == 0` both operands are null pointer constants. The C90 wording permits the left operand to be converted to the type of the right operand (type **int**). The C99 wording does not support this interpretation. 748 null pointer constant

C++

The C++ Standard supports this combination of operands but does not explicitly specify any sequence of operations that take place prior to the comparison. 1217 equality operators null pointer constant

Other Languages

Other languages do not always go into this level of detail. They usually simply specify that two null pointers compare equal, irrespective of the pointer types involved.

Coding Guidelines

The issue of using explicit casts in this case is discussed elsewhere. 1216 equality operators pointer to incomplete type

1231 If one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter. equality operators pointer to void

Commentary

A pointer to **void** is capable of representing all the information represented in any pointer type (any pointer converted to pointer to **void** and back again will compare equal to the original pointer). There is no guarantee that any other pointer type will be capable of representing all the information in the pointer to **void** (although pointer to character types have the same representation and alignment requirements). For this reason the other operand has to be converted to pointer to **void**.

pointer 744 converted to pointer to void
pointer 558 to void same representation and alignment as

C++

This conversion is part of the general pointer conversion (4.10) rules in C++. This conversion occurs when two operands have pointer type.

Coding Guidelines

The issue of using explicit casts in this case is discussed elsewhere.

equality operators 1216 pointer to incomplete type

---

equality operators pointer to object

For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. 1232

Commentary

This wording was added by the response to DR #215 (the Committee response was to duplicate the wording from relational operators). The same sentence appears elsewhere in the standard and the issues are discussed there.

relational operators 1203 pointer to object
additive operators 1165 pointer to object

---

pointers compare equal

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[92)] 1233

Commentary

This is a requirement on the implementation. If the expression `px==py` returns 1 (with `px` and `py` both having pointer types), one of the conditions listed here must hold (the fact that two pointer values compare equal does not imply that they have the same value representation). In all other cases an implementation is required to return 0. All null pointers, whatever type they have been converted to, always compare equal. Here the phrase " . . . point to the same object . . . " means the same address.

pointer 590 segmented architecture
null pointer 750 conversion yields null pointer
object 761 lowest addressed byte
pointer 1169 one past end of object

The standard requires that pointers be able to point one past the last element of an array, but it does not place any requirements on the relative storage addresses of different objects. This can lead to the situation of a one past the last element pointer comparing equal to a pointer to a different object (which might not even be compatible with the pointed-to type). The standard does not require such behavior, it simply points out that it can occur in a strictly conforming program.

pointer subtraction 1173 point at same object

The relationship `p == q` does not imply `(p-q) == 0`, because the subtraction operator is only defined when its operands point at an object. If `p` and `q` have the null pointer value, the result is undefined behavior. The following equalities are true if `p != NULL`, but are not guaranteed to be true if `p == NULL` (although in practice they are always true on most implementations);

```
1  p - p == 0
2  p + 0 == p
```

C90

If two pointers to object or incomplete types are both null pointers, they compare equal. If two pointers to object or incomplete types compare equal, they both are null pointers, or both point to the same object, or both point

one past the last element of the same array object. If two pointers to function types are both null pointers or both point to the same function, they compare equal. If two pointers to function types compare equal, either both are null pointers, or both point to the same function.

The admission that a pointer one past the end of an object and a pointer to the start of a different object compare equal, if the implementation places the latter immediately following the former in the address space, is new in C99 (but it does describe the behavior of most C90 implementations).

C++

Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array. 5.10p1

This specification does not include the cases:

- "(including a pointer to an object and a subobject at its beginning)", which might be deduced from wording given elsewhere,
- "or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space". The C++ Standard does not prevent an implementation from returning a result of **`true`** for the second case, but it does not require it. However, the response to C++ DR #073 deals with the possibility of a pointer pointing one past the end of an object comparing equal, in some implementations, to the address of another object. Wording changes are proposed that acknowledge this possibility.

761 object
lowest addressed byte

Other Languages

These requirements usually hold for implementations of other languages, but are rarely expressed explicitly.

Common Implementations

If two pointers point to the same object, and the lifetime of that object ends, most implementations will continue to compare those pointers as being equal, even though they no longer point at an object. If an integer value is cast to a pointer type and assigned to two pointers, most implementations will compare those pointers as equal, even though neither of them may be the null pointer constant or have ever pointed at an object.

6.5.10 Bitwise AND operator

1234 AND-expression
syntax
bitwise
&

```
AND-expression:
                equality-expression
                AND-expression & equality-expression
```

Commentary

From decvax!harpo!npoiv!alice!research!dmr Fri Oct 22 01:04:10 1982 Subject: Operator precedence Newsgroups: net.lang.c Dennis Ritchie

The priorities of && || vs. == etc. came about in the following way.

Early C had no separate operators for & and && or | and ||. (Got that?) Instead it used the notion (inherited from B and BCPL) of "truth-value context": where a Boolean value was expected, after "if" and "while" and so forth, the & and | operators were interpreted as && and || are now; in ordinary expressions, the bitwise

interpretations were used. It worked out pretty well, but was hard to explain. (There was the notion of "top-level operators" in a truth-value context.)

The precedence of & and | were as they are now.

Primarily at the urging of Alan Snyder, the && and || operators were added. This successfully separated the concepts of bitwise operations and short-circuit Boolean evaluation. However, I had cold feet about the precedence problems. For example, there were lots of programs with things like

if (a==b & c==d) . . .

In retrospect it would have been better to go ahead and change the precedence of & to higher than ==, but it seemed safer just to split & and && without moving & past an existing operator. (After all, we had several hundred kilobytes of source code, and maybe 3 installations. . . .)

Other Languages

Languages that support bitwise operators tend to either use the same operator as C, or the keyword **and**. Support for an unsigned integer type was added in Ada 95 and the definition of the logical operators was extended to perform bitwise operations when their operands had this type.

Coding Guidelines

logical-AND-expression 1248 syntax

The issue of rearranging expressions, involving bitwise operators, to reduce the number of operators they contain is discussed elsewhere.

controlling expression 1740 if statement

Experience shows that developers sometimes confuse the two operators **&** and **&&** with each other, and sometimes they intentionally swap the use of these operators. The confusion may be caused by their similar visual appearance, a temporary lapse of concentration, or some other reason. They may return different results for the same pair of operands, as the following example shows:

```
0x10 & 0x01  ⇒ 0
0x10 && 0x01 ⇒ 1
```

The two operators return the same numeric result when the evaluation of the second operand generates no side effects, and

- the value of either operand is 0,
- boolean role 476 the values of both operands is restricted to the range 0 or 1 (a boolean role represented as one of two values), and
- the least significant bit of both operands is set and the operands have no other set bits in common.

When there is no order dependency between the operands (i.e., the evaluation of the second operand is not conditional on the evaluation of the first), developers sometimes make use of one of these equivalencies to select what they consider to be the *most efficient* operator. For instance, if the one of the operands is a function call, **&&** might be used with the call as its right operand. Alternatively, if both operands are simple object accesses, the **&** operator might be used in the belief that unconditionally evaluating both operands is faster than a conditional evaluation (because of the conditional jump).

There are two checks that might be used to attempt to confirm that the operator appearing in the source is the one intended:

object 1352 role, boolean role 476, bit-set role 945, numeric role 945

1. *The roles of the operands.* If both operands have boolean roles, the operators **&** and **&&** both return the same result and the author is free to select which operator to use. Operands having a bit-set role might be expected to be operands of the **&** operator, while those having a numeric role appear as operands of the **&&** operator.

2. *The role played by the result.* The result of the **&** operator has a bit-set role, while that of the **&&** operator has a boolean role. For instance, the measurements in Table 1234.1 show that most occurrences of the **&&** operator are in a controlling expression (both are boolean roles).

The issue of operands having the same role is covered by the guideline recommendation dealing with objects having a single role (the definitions of bit-set and numeric roles are based on objects appearing as operands of certain kinds of operators. An object that appeared as the operand of both kinds of operator would have two roles).

role
operand matching
1352.1 object
used in a single role
945 bit-set role
945 numeric role

Other coding guideline documents sometimes specify that these two operators should not be confused, or list them in review guidelines.[1497] Using **&&** where **&** was intended, or vice versa, is clearly unintended (a fault) and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. However, it may be possible to reduce the likelihood of confusion by using the operator appropriate to the role. The issue of the role of operands matching that of their operators is discussed elsewhere.

0 MISRA
0 guidelines
not faults
1234 role
operand matching

Table 1234.1: Occurrence of the **&** and **&&** operator (as a percentage of all occurrences of each operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	Binary &	&&
if control-expression	51.4 (10.5)	82.4 (10.4)
other contexts	45.3 (—)	7.7 (—)
while control-expression	2.1 (8.1)	6.9 (18.4)
for control-expression	0.3 (0.6)	3.0 (4.7)
switch control-expression	0.8 (5.2)	0.0 (0.0)

Table 1234.2: Common token pairs involving one of the operators **&**, **|**, or **^** (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier **|**	0.4	74.0	**&** identifier	57.1	0.6
identifier **&**	0.7	67.5	**|** identifier	79.8	0.4
identifier **^**	0.0	51.1	**& (**	7.4	0.3
) ^	0.0	38.7	**| (**	14.4	0.3
& ~	4.6	30.1	**^ *v**	5.5	0.1
) &	1.1	27.7	**|** *integer-constant*	5.5	0.1
) |	0.4	20.8	**^** *integer-constant*	20.8	0.0
] ^	0.0	5.1	**^** identifier	55.5	0.0
] &	1.4	4.2	**^ (**	16.1	0.0
& *integer-constant*	30.6	1.5			

Constraints

1235 Each of the operands shall have integer type.

& binary
operand type

Commentary

This constraint reflects the fact that processors very rarely contain instructions for performing this operation on non-integer types. In turn this reflects the fact that there is no commonly defined semantics for bitwise ANDing other types and that there are very few algorithms requiring values having other types to be treated as a sequence of bits.

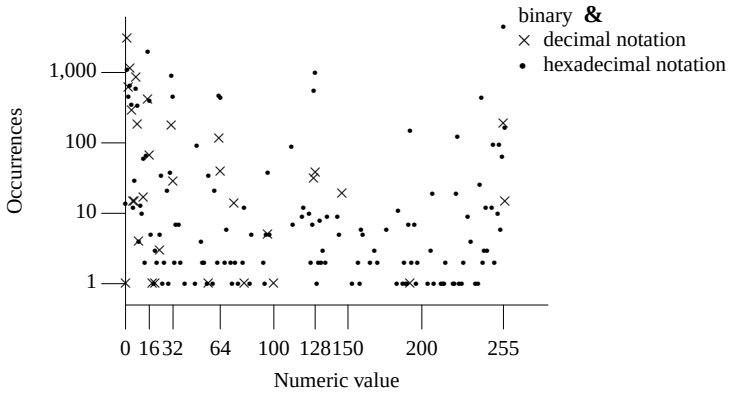


Figure 1234.1: Number of *integer-constant*s having a given value appearing as the right operand of the binary **&** operator. Based on the visible form of the `.c` files.

C++

The wording of the specification in the C++ Standard is somewhat informal (the same wording is given for the bitwise exclusive-OR operator, 5.12p1, and the bitwise inclusive-OR operator, 5.13p1).

5.11p1 *The operator applies only to integral or enumeration operands.*

Coding Guidelines

representation information using boolean role 476 bit-set role 945

The binary **&** operator operates on the bit representation of its operands. As such, the guideline recommendation on making use of representation information is applicable. However, deviations are suggested for operands having a boolean or bit-set role.

Table 1235.1: Occurrence of bitwise operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	|	**_int**	27.1	**unsigned int**	|	**unsigned int**	4.0
int	**&**	**_int**	24.3	**unsigned long**	**&**	**_int**	3.8
_int	|	**_int**	23.0	**unsigned int**	|	**unsigned long**	3.4
unsigned int	^	**unsigned int**	17.7	**unsigned int**	^	**_int**	3.3
other-types	**&**	other-types	13.9	**unsigned int**	^	**int**	3.1
int	|	**int**	13.6	**unsigned long**	**&**	**int**	2.6
_int	^	**_int**	13.5	**long**	^	**long**	2.6
unsigned long	^	**unsigned long**	12.2	**unsigned char**	**&**	**int**	2.5
unsigned int	**&**	**_int**	11.5	**unsigned long**	|	**unsigned long**	2.4
unsigned char	**&**	**_int**	10.3	**unsigned long**	**&**	**unsigned long**	2.0
int	^	**_int**	10.3	**unsigned int**	^	**unsigned char**	1.8
other-types	^	other-types	9.9	**unsigned short**	^	**unsigned short**	1.7
int	^	**int**	9.8	**int**	^	**unsigned char**	1.7
unsigned int	|	**int**	9.6	**unsigned short**	**&**	**unsigned short**	1.5
other-types	|	other-types	8.9	**unsigned short**	^	**_int**	1.5
unsigned short	**&**	**_int**	7.1	**long**	**&**	**int**	1.4
int	**&**	**int**	6.3	**int**	|	**unsigned char**	1.4
unsigned int	**&**	**int**	5.7	**unsigned short**	**&**	**int**	1.3
long	|	**long**	5.5	**unsigned int**	^	**unsigned short**	1.3
unsigned int	**&**	**unsigned int**	4.6	**long**	**&**	**_int**	1.2
unsigned char	^	**unsigned char**	4.6	**_int**	|	**int**	1.2
unsigned char	^	**_int**	4.2	**int**	^	**unsigned short**	1.1

Semantics

1236 The usual arithmetic conversions are performed on the operands.

& binary operands converted

Commentary

The rationale for performing these conversions is a general one that is not limited to the operands of the arithmetic operators.

702 operators cause conversions

C++

The following conversion is presumably performed on the operands.

The usual arithmetic conversions are performed; 5.11p1

Common Implementations

If one of the operands is positive and has a type with lower rank than the other operand, it may be possible to make use of processor instructions that operate on narrower width values. (Converting the operand to the greater rank will cause it to be zero extended, which will cancel out any ones in the other operand.) Unless both operands are known to be positive, there tend to be few opportunities for optimizing occurrences of the `^` and `|` operators (because of the possibility that the result may be affected by an increase in value representation bits).

Coding Guidelines

Unless both operands have the same type, which also has a rank at least equal to that of **`int`**, these conversions will increase the number of value representation bits in one or both operands. Given that the binary **`&`** operator is defined to work at the bit level, developers have to invest additional effort in considering the effects of the usual arithmetic operands on the result of this operator.

A probabilistic argument could be used to argue that of all the bitwise operators the **`&`** operator has the lowest probability of causing a fault through an unexpected increase in the number of value representation bits. For instance, the probability of both operands having a negative value (needed for any additional bits to be set in the result) is lower than the probability of one operand having a negative value (needed for a bit to be set in the result of the `^` and `|` operators).

Unless the operands have a bit-set role, the guideline recommendation dealing with use of representation information is applicable here.

945 bit-set role 569.1 representation information using

1237 The result of the binary `&` operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

Commentary

This information is usually expressed in tabular form.

	0	1
0	0	0
1	0	1

Common Implementations

The Unisys A Series[1390] uses signed magnitude representation. If the operands have an unsigned type, the bit used to represent the sign in signed types, which is present in the object representation on this processor, does not take part in the binary **`&`** operation. If the operands have a signed type, the sign bit does take part in the bitwise-AND operation.

Coding Guidelines

Although the result of the bitwise-AND operator is the common type derived from the usual arithmetic conversions, for the purpose of these guideline recommendations its role is the same as that of its operands.

706 usual arithmetic conversions 1352 object role

footnote 92

92) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. 1238

Commentary

array 526
contiguously allocated set of objects
structure 1206
members
later compare later
structure 1424
unnamed padding
storage 1354
layout

By definition elements of an array are contiguous and the standard specifies the relative order of members and their possible padding. Some implementations treat objects defined by different declarations in the same way as the declaration of members in a structure definition. For instance, assigning the same relative offsets to objects local to a function definition as they would the equivalent member declarations in a structure type. The issue of the layout of objects in storage is discussed elsewhere.

This footnote lists all the cases where objects may be adjacent in memory.

C90

The C90 Standard did not discuss these object layout possibilities.

C++

The C++ Standard does not make these observations.

Other Languages

Most languages do not get involved in discussing this level of detail.

Common Implementations

Most implementations aim to minimize the amount of padding between objects. In many cases objects defined in block scope are adjacent in memory to objects defined textually adjacent to them in the source code.

If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior. 1239

Commentary

This describes a special case of undefined behavior. (It is called out because, in practice, it is more likely to occur for values having pointer types than values having integer types.) Once undefined behavior has occurred, any subsequent operation can also produce undefined behavior. The standard does not limit the scope of undefined behaviors to operations involving operands that cause the initial occurrence.

Common Implementations

On many implementations the undefined behavior that occurs when additive operations, on values having an integer type, overflow is *symmetrical*. That is, an overflow in one direction can be undone by an overflow in the other direction (e.g., `INT_MAX+2-2` produces the same result as `INT_MAX-2+2`). On implementations for processors using a segmented architecture this symmetrical behavior may not occur, for values having pointer types, because of internal details of how pointer arithmetic is handled on segment boundaries.

pointer 590
segmented architecture
arithmetic 687
saturated

On some processors additive operations, on integer or pointer types, saturate. In this case the undefined behavior is not symmetrical.

C90

The C90 Standard did not discuss this particular case of undefined behavior.

6.5.11 Bitwise exclusive OR operator

exclusive-OR-expression: 1240
AND-expression
exclusive-OR-expression ^ *AND-expression*

Commentary

The ^ operator can be replaced by a sequence of equivalent bitwise operators; for instance, `x ^ y` can be written `(x & (~y)) | ((~x) & y)`. However, most processors contain a bitwise exclusive-OR instruction and thus this operator is included in C. Being able to represent a bitwise operator using an equivalent sequence of other operators is not necessarily an argument against that operator being included in C. It is possible to represent any boolean expression using a sequence of NAND (*not and*, e.g., `!(x & y)`) operators; for instance, `!x` becomes `x NAND x`, `x & y` becomes `(x NAND y) NAND (x NAND y)`, and so on. Although this equivalence may be of practical use in hardware design (where use of mass-produced circuits performing a single boolean operation can reduce costs), it is not applicable to software.

Other Languages

Languages that support bitwise operations usually support an exclusive-OR operator. The keyword **xor** is sometimes used to denote this operator. The ^ character is used as a token to indicate pointer indirection in Pascal and Ada.

Coding Guidelines

The exclusive-OR operator is not encountered in everyday life. There is no English word or phrase that expresses its semantics. The everyday usage discussed in the subclause on the logical-OR operator is based on selecting between two alternatives (e.g., *either one or the other* which only deals with two of the four possible combinations of operand values). Because of the lack of everyday usage, and because it does not occur often within C source (compared with bitwise-AND and bitwise-OR (see Table 912.2)) it is to be expected that developers will have to expend more effort in comprehending the consequences of this operator when it is encountered in source code.

1256 logical-OR-expression syntax

The greater cognitive cost associated with use of the exclusive-OR operator is not sufficient to recommend against its use. Developers need to be able to make use of an alternative that has a lower cost. While the behavior of the exclusive-OR operator can be obtained by various combinations of other bitwise operators, it seems unlikely that the cost of comprehending any of these sequences of operators will be less than that of the operator they replace. If the exclusive-OR operator appears within a more complicated boolean expression it may be possible to rewrite that expression in an alternative form. Rewriting an expression can increase the cognitive effort needed for readers to map between source code and the application domain (i.e., if the conditions in the application domain are naturally expressed in terms of a sequence of operators that include exclusive-OR). The cost/benefit of performing such a rewrite needs to be considered on a case by case basis.

1248.1 boolean expression minimize effort

Example

The ^ operator is equivalent to the binary + operator if its operands do not have any set bits in common. This property can be used to perform simultaneous addition on eight digits represented in packed BCD[687] (i.e., four bits per digit).

```
int BCD_add(a, b)
{
int t1 = a + 0x06666666,
    t2 = t1 + b,
    t3 = t1 ^ b,
    t4 = t2 ^ t3,
    t5 = ~t4 & 0x11111110,
    t6 = (t5 >> 2) | (t5 >> 3);
return t2 - t6;
}
```

Usage

The ^ operator represents 1.2% of all occurrences of bitwise operators in the visible source of the `.c` files.

Constraints

1241 Each of the operands shall have integer type.

Commentary

& binary operand type 1235

The discussion for the various subsections is the same as those for the bitwise AND operator.

Semantics

---

^ operands converted

The usual arithmetic conversions are performed on the operands. 1242

Commentary

& binary operands converted 1236

The discussion in the various subsections is the same as that for the bitwise AND operator.

The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set). 1243

Commentary

This information is usually expressed in tabular form.

	0	1
0	0	1
1	1	0

Common Implementations

The Unisys A Series[1390] uses signed magnitude representation. If the operands have an unsigned type, the sign bit is not affected by the bitwise complement operator. If the operands have a signed type, the sign bit does take part in the bitwise complement operation.

The bitwise exclusive-OR instruction is sometimes generated, by optimizers, to swap the contents of two registers, without using a temporary register (as shown in the Example below).

Coding Guidelines

usual arithmetic conversions object role 706 1352

Although the result of the bitwise exclusive-OR operator is the common type derived from the usual arithmetic conversions, for the purpose of these guideline recommendations its role is the same as that of its operands.

Example

```
1  #define SWAP(x, y) (x=(x ^ y), y=(x ^ y), x=(x ^ y))
2  #define UNDEFINED_SWAP(x, y) (x ^= y ^= x ^= y) /* Requires right to left evaluation. */
```

6.5.12 Bitwise inclusive OR operator

inclusive-OR-expression syntax

1244

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

Commentary

AND-expression syntax 1234

The discussion on AND-expression is applicable here.

Coding Guidelines

The | and || operators differ from their corresponding AND operators in that the zero/nonzero status of their result is always the same, even though the actual result values are likely to differ.

```
0x10 | 0x01  ⇒ 0x11
0x10 || 0x01 ⇒ 1
```

While it is possible to use either operators in a context where the result is compared against zero, the guideline recommendation dealing with matching an operator to the role of its result still applies. The pattern of operator context usage (see Table 1244.1) is similar to that of the two AND operators.

1234 role operand matching

Table 1244.1: Occurrence of the | and || operator (as a percentage of all occurrences of each operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	|	||
if control-expression	8.8 (0.7)	86.0 (6.9)
other contexts	90.7 (—)	11.9 (—)
while control-expression	0.3 (0.5)	1.9 (2.7)
for control-expression	0.0 (0.0)	0.3 (0.2)
switch control-expression	0.1 (0.3)	0.0 (0.0)

Constraints

1245 Each of the operands shall have integer type.

Commentary

The discussion for the various subsections is the same as those for the bitwise AND operator.

1235 & binary operand type

Semantics

1246 The usual arithmetic conversions are performed on the operands.

Commentary

The discussion in the various subsections is the same as that for the bitwise exclusive-OR operator.

1242 ^ operands converted

1247 The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

Commentary

This information is usually expressed in tabular form.

	0	1
0	0	1
1	1	1

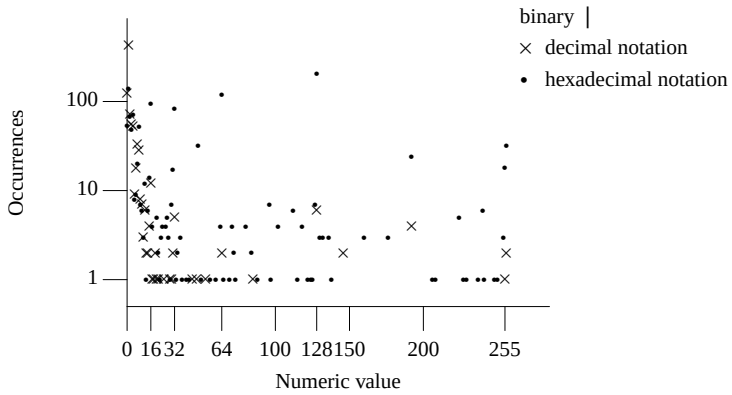


Figure 1244.1: Number of *integer-constant*s having a given value appearing as the right operand of the bitwise-OR operator. Based on the visible form of the `.c` files.

Common Implementations

The Unisys A Series[1390] uses signed magnitude representation. If the operands have an unsigned type, the sign bit is not affected by the bitwise-OR operator. If the operands have a signed type, the sign bit does take part in the bitwise-OR operation.

Coding Guidelines

usual arithmetic conversions 706
object 1352
role

Although the result of the bitwise inclusive-OR operator is the common type derived from the usual arithmetic conversions, for the purpose of these guideline recommendations its role is the same as that of its operands.

6.5.13 Logical AND operator

logical-AND-expression syntax

1248

```
logical-AND-expression:
                inclusive-OR-expression
                logical-AND-expression && inclusive-OR-expression
```

Commentary

The relative precedence of the binary **&&** and **||** operators is the same as that of the binary **&** and **|** operators.

Other Languages

In many languages the logical-AND operator has higher precedence than the logical-OR operator. In a few languages (Ada, Fortran which uses the keyword **.AND.**), they have the same precedence. Ada supports two kinds of logical-AND operations: **and** and **and then**, the latter having the same semantics as the logical-AND operator in C (short-circuit evaluation). These operators also have the same precedence as the logical-OR and logical-XOR operators.

bitwise & 1234

Coding Guidelines

AND-expression syntax 1234

The issue of swapping usages of the **&** and **&&** operators is discussed elsewhere.

There are a number of techniques for simplifying expressions involving boolean values. One of the most commonly used methods is the Karnaugh map.[714] (For equations involving more than five operands, the Quine-McCluskey technique[1357] may be more applicable; this technique is also capable of being automated.), while some people prefer algebraic manipulation (refer to Table 1248.1).

Although simplification may lead to an expression that requires less reader effort to comprehend as a boolean expression, the resulting expression may require more effort to map to the model of the application domain being used. For instance, `(A && (!B)) || ((!A) && B)` can be simplified to `A ^ B` (assuming `A` and `B` only take the values 0 and 1, otherwise another operation is required). However, while the use of the exclusive-OR operator results in a visually simpler expression, developers have much less experience dealing with it than the other bitwise and logical operators. There is also the possibility that, for instance, the expression `(A && (!B))` occurs in other places within the source and has an easily-deduced meaning within the framework of the application model.

mind logical operator

Logical operators are part of mathematical logic. Does the human mind contain special circuitry that performs this operation, just like most processors contain a group of transistors that perform this C operation? Based on experimental observations, the answer would appear to be no. So how does the human mind handle the logical-AND operation? One proposal is that people learn the rules of this operator by rote so that they can later be retrieved from memory. The result of a logical operation is then obtained by evaluating each operand's condition to true or false and performing a table lookup using previously the learned logical-AND table to find the result. Such an approach relies on short-term memory, and the limited capacity of short-term memory offers one explanation of why people are poor at evaluating moderately complex logical operators in their head. There is insufficient capacity to hold the values of the operands and the intermediate results.

The form of logical deduction that is needed in comprehending software rarely occurs in everyday life. People's ability to solve what appear to be problems in logic does not mean that the methods of boolean mathematics are used. A number of other proposals have been made for how people handle *logical* problems

in everyday life. One is that the answers to the problems are simply remembered; after many years of life experience, people accumulate a store of knowledge on how to deal with different situations.

Studies have found that belief in a particular statement being true can cause people to ignore its actual status as a mathematical expression (i.e., people believe what they consider to be true rather than logically evaluating the true status of an expression). Estimating developers' performance at handling logical expressions therefore involves more than a model of how they process mathematical logic.

While minimizing the number of operands in a boolean expression may have appeal to mathematically oriented developers, the result may be an expression that requires more effort to comprehend. It is not yet possible to calculate the reader effort required to comprehend a particular boolean expression. For this reason the following guideline recommendation relies on the judgment of those performing the code review.

Rev 1248.1

Boolean expressions shall be written in a form that helps minimize the effort needed by readers to comprehend them.

A comment associated with the simplified expression can be notated in at least two ways: using equations or by displaying the logic table. Few developers are sufficiently practiced at boolean algebra to be able to fluently manipulate expressions; a lot of thought is usually required. A logic table highlights all combinations of operands and, for small numbers of inputs, is easily accommodated in a comment.

Table 1248.1: Various identities in boolean algebra expressed using the `||` and **`&&`** operators. Use of these identities may change the number of times a particular expression is evaluated (which is sometimes the rationale for rewriting it). The relative order in which expressions are evaluated may also change (e.g., when `A==1` and `B==0` in `(A && B) || (A && C)` the order of evaluation is A⇒ B⇒ A⇒ C, but after use of the distributive law the order becomes A⇒ B⇒ C).

Relative Order Preserved	Expression ⇒ Alternative Representation
	Distributive laws
no	`(A && B) || (A && C)` ⇒ `A && (B || C)`
no	`(A || B) && (A || C)` ⇒ `A || (B && C)`
	DeMorgan's theorem
yes	`!(A || B)` ⇒ `(!A) && (!B)`
yes	`!(A && B)` ⇒ `(!A) || (!B)`
	Other identities
yes	`A && ((!A) || B)` ⇒ `A && B`
yes	`A || ((!A) && B)` ⇒ `A || B`
	The consensus identities
no	`(A && B) || ((!A) && C) || (B && C)` ⇒ `(A && B) || ((!A) && C))`
yes	`(A && B) || (A && (!B) && C)` ⇒ `(A && B) || (A && C)`
yes	`(A && B) || ((!A) && C)` ⇒ `((!A) || B) && (A || C)`

An expression containing a number of logical operators, each having operands whose evaluation involves relational or equality operators, can always be written in a number of different ways, for instance:

```
1  if ((X < 4) && !(Y || (Z == 1)))
2     /* ... */
3
4  if ((Y != 0) && (Z != 0) && (X < 4))
5     /* ... */
6
7  if (!((X >= 4) || Y || (Z == 1)))
8     /* ... */
9
10 if (X < 4)
11    if (!(Y || (Z == 1)))
12       /* ... */
```

An example of complexity in an English sentence might be "Suppose five days after the day before yesterday is Friday. What day of the week is tomorrow?" Whether the use of a less complex (i.e., having less cognitive load) expression has greater cost/benefit than explicitly calling out the details of the calculation needs to be determined on a case-by-case basis.

categorization performance predicting 0

A study by Feldman[414] found that the subjective difficulty of a concept (e.g., classifying colored polygons of various sizes) was directly proportional to its boolean complexity (i.e., the length of the shortest logically equivalent propositional formula).

Table 1248.2: Common token pairs involving **&&**, or **||** (as a percentage of all occurrences of each token). Based on the visible form of the **.c** files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier **&&**	0.4	48.5	**&&** defined	0.9	6.2
) **||**	0.9	42.7	**||** **!**	11.3	6.0
identifier **||**	0.2	39.3	*character-constant* **||**	4.2	4.2
) **&&**	1.1	34.9	*character-constant* **&&**	5.3	3.3
|| defined	4.8	21.0	**&&** **(**	28.7	0.9
integer-constant **||**	0.3	12.4	**||** **(**	29.7	0.6
integer-constant **&&**	0.4	11.5	**&&** identifier	53.9	0.5
&& **!**	13.5	11.3	**||** identifier	51.8	0.3

Constraints

&& operand type

Each of the operands shall have scalar type. 1249

Commentary

The behavior is defined in terms of an implicit comparison against zero, an operation which is only defined for operands having a scalar type in C.

C++

5.14p1 *The operands are both implicitly converted to type* ***bool*** *(clause 4).*

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Other Languages

Languages that support boolean types usually require that the operands to their logical-AND operator have boolean type.

Coding Guidelines

& binary 1235 operand type if statement 1743 controlling expression scalar type

The discussion on the bitwise-AND operator is also applicable here, and the discussion on the comprehension of the controlling expression in an **if** statement is applicable to both operands.

Table 1249.1: Occurrence of logical operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	**||**	**int**	87.7	**_long**	**||**	**_long**	2.2
int	**&&**	**int**	73.9	**int**	**&&**	ptr-to	2.2
other-types	**&&**	other-types	12.8	**int**	**&&**	**char**	1.8
other-types	**||**	other-types	8.4	**int**	**||**	**_long**	1.7
ptr-to	**&&**	**int**	4.5	**int**	**&&**	**_int**	1.3
char	**&&**	**int**	2.3	ptr-to	**&&**	ptr-to	1.1

Semantics

1250 The **&&** operator shall yield 1 if both of its operands compare unequal to 0;

&& operand compare against 0

Commentary

The only relationship between the two operands is their appearance together in a logical-AND operation. There is no benefit, from the implementations point of view, in performing the usual arithmetic conversions or the integer promotions on each operand.

It is sometimes necessary to test a preliminary condition before the main condition can be tested. For instance, it may be necessary to check that a pointer value is not null before dereferencing it. The test could be performed using nested **if** statements, as in:

```
1   if (px != NULL)
2      if (px->m1 == 1)
```

More complex conditions are likely to involve the creation of a warren of nested **if** statements. The original designers of the C language decided that it was worthwhile creating operators to provide a shorthand notation (i.e., the logical-AND and logical-OR operators). In the preceding case use of one of these operators allows both tests to be performed within a single conditional expression (e.g., `if ((px != NULL) && (px->m1 == 1))`). The other advantage of this operator, over nested **if** statements, is in the creation of expressions via the expansion of nested macro invocations. Generating nested **if** statements requires the use of braces to ensure that any following **else** arms are associated with the intended **if** arm. Generating these braces introduces additional complexities (at least another macro invocation in the source) that don't occur when the **&&** operator is used.

C++

The result is ***true*** *if both operands are* ***true*** *and* ***false*** *otherwise.* 5.14p1

The difference in operand types is not applicable because C++ defines equality to return **true** or **false**. The difference in return value will not cause different behavior because **false** and **true** will be converted to 0 and 1 when required.

Other Languages

The mathematical use of this operator returns true if both operands are true. Languages that support a boolean data type usually follow this convention.

Common Implementations

As discussed elsewhere, loading a value into a register often sets conditional flags as does a relational operation comparing two values. This means that in many cases machine code to compare against zero need not be generated, as in, the following condition:

1111 logical negation result is 1210 relational operators result value

```
1   if ((a == b) && (c < d))
```

which is likely to generate an instruction sequence of the form:

```
                compare a with b
                if not equal jump to else_or_endif
                compare c with d
                if greater than or equal to jump to else_or_endif
                code for if arm
else_or_endif:  rest of program
```

Coding Guidelines

Should the operands always be explicitly compared against zero? When an operand is the result of a relational or equality operator, such a comparison is likely to look very unusual:

```
1   if (((a == b) == 0) && ((c < d) == 0))
```

It is assumed that developers think about the operands in terms of boolean roles and the result of both the relational and equality operators have a boolean role. In these cases another equality comparison is redundant. When an operand does not have a boolean role, an explicit comparison against zero might be appropriate (these issues are also discussed elsewhere).

! 1113 equivalent to selection statement 1739 syntax

---

otherwise, it yields 0. 1251

Commentary

That is, a value of zero having type **int**.

&& result type

The result has type **int**. 1252

Commentary

relational 1211 operators result type

The rationale for this choice is the same as for relational operators.

C++

5.14p2 *The result is a **bool**.*

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

Other Languages

In languages that support a boolean type, the result of logical operators usually has a boolean type.

Common Implementations

If the result is immediately cast to another type, an implementation may choose to arrange that other type as the result type; for instance, in:

```
1   float f(int ip)
2   {
3   return (ip > 0) && (ip < 10);
4   }
```

an implementation may choose to return `0.0` and `1.0` as the result of the expression evaluation, saving a cast operation.

Coding Guidelines

relational 1211 operators result type

The coding guideline issues are the same as those for the relational operators.

&& evaluation order

Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; 1253

Commentary

The left-to-right evaluation order of the **&&** operator is required; it is what makes it possible for the left operand to verify a precondition for the evaluation of the right operand. There is no requirement that the evaluation of the right operand, if it occurs, take place immediately after the evaluation of the left operand.

Coding Guidelines

sequence 187.1 points all orderings give same value

This issue is covered by the guideline recommendation dealing with sequence points.

Example

In the following:

```
#include <stdio.h>

void f(void)
{
(printf("Hello ") && printf("World\n")) + printf("Goodbye\n");
}
```

possible output includes:

```
Hello World
Goodbye
```

and

```
Hello Goodbye
World
```

1254 there is a sequence point after the evaluation of the first operand.

&&
sequence point

Commentary

Unlike the nested **if** statement form, there is no guaranteed sequence after the evaluation of the second operand, if it occurs.

1250 && operand compare against 0

C++

All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.14p2

The possible difference in behavior is the same as for the function-call operator.

1025 function call sequence point

Coding Guidelines

While the sequence point ensures that any side effects in the first operand have completed, relying on this occurring creates a dependency within the expression that increases the effort needed to comprehend it. Some of the issues involving side effects in expressions are discussed elsewhere.

941 object modified once between sequence points

Example

```
#include <stdio.h>

extern int glob;

void f(void)
{
if (--glob && ++glob)
   printf("The value of glob is neither 0 nor 1\n");
}
```

1255 If the first operand compares equal to 0, the second operand is not evaluated.

&&
second operand

Commentary

An alternative way of looking at this operator is that `x && y` is equivalent to `x ? (y?1:0) : 0`.

Common Implementations

This operand can be implemented, from the machine code generation point of view, as if a nested **`if`** construct had been written.

Coding Guidelines

Some coding guideline documents prohibit the second operand containing side effects. The rationale is that readers of the source may fail to notice that if the second operand is not evaluated any side effects it generates will not occur. The majority of C's binary operators (32 out of 34) always evaluate both of their operands. Do developers make the incorrect assumption that the operands of all binary operators are always evaluated? Your author thinks not. Many developers are aware that in some cases the **`&&`** operator is more efficient than the **`&`** operator, because it only evaluates the second operand if the first is not sufficient to return a result.

Although many developers may be aware of the conditional evaluation of the second operand, some will believe that both operands are evaluated. While a guideline recommendation against side effects in the second operand may have a benefit for some developers, it potentially increases cost in that the alternative construct used may require more effort to comprehend (e.g., the alternative described before). Given that the alternative constructs that could be used are likely to require more effort to comprehend and the unknown percentage of developers making incorrect assumptions about the evaluation of the operands, no guideline recommendation is given.

coverage testing

Coverage testing

There are software coverage testing requirements that are specific to logical operators. Branch condition decision testing involves the individual operands of logical operators. It requires that test cases be written to exercise all combinations of operands. In:

```
1  if (A || (B && C))
```

it is necessary to verify that all combinations of `A`, `B`, and `C`, are evaluated. In the case of the condition involving `A`, `B`, and `C` eight separate test cases are needed. For more complex conditions, the number of test cases rapidly becomes impractical.

Modified condition decision testing requires that test cases be written to demonstrate that each operand can independently affect the output of the decision. For the **`if`** statement above, Table 1255.1 shows the possible conditions. Using modified condition, *MC*, coverage can significantly reduce the number of test cases over branch condition, *BC*, coverage.

Table 1255.1: Truth table showing how each operand of `(A || (B && C))` can affect its result. Case 1 and 2 show that `A` affects the outcome; Case 2 and 3 shows that `B` affects the outcome; Case 3 and 4 shows that `C` affects the outcome.

Case	A	B	C	Result
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

The FAA requires[1183] that aviation software at Level A (the most critical, defined as that which could prevent continued safe flight and landing of the aircraft) have the level of coverage specified by MC/DC.[104] This requirement has been criticized as not having a worthwhile cost (money and time) and benefit (the number of errors found) ratio. An empirical evaluation of the HETE–2 satellite software[372] looked at the faults found by various test methods and their costs. Logical operators can also appear within expressions that are not a condition context. The preceding coverage requirements also hold in these cases.

Horgan and London[593] describe an Open Source tool that measures various dataflow coverage metrics for C source.

6.5.14 Logical OR operator

1256 logical-OR-expression syntax

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

Commentary

The discussion on the logical-AND operator is applicable here. 1248 logical-AND-expression syntax

Other Languages

Ada supports two kinds of logical-OR operations: **`or`** and **`or else`**. The latter has the same semantics as the logical-OR operator in C (short-circuit evaluation). Support for an unsigned integer type was added in Ada 95 and the definition of the logical operators was extended to perform bitwise operations when their operands had this type.

Coding Guidelines

The issue of swapping usages of the `|` and `||` operators is discussed elsewhere. 1244 inclusive-OR-expression syntax

In English (and other languages) the word *or* is sometimes treated as having an exclusive rather than an inclusive meaning. For instance, "John has an M.D. or a Ph.D." is likely to be interpreted in the exclusive sense, while "John did not buy ice cream or chips" is likely to be interpreted in the inclusive sense (e.g., John did not buy ice cream and did not buy chips); the exclusive sense supports surprising possibilities (e.g., John buying ice cream and buying chips). A number of studies[403] have found that people make more mistakes when answering questions that involve disjunctive (i.e., the logical-OR operator) relationships than when answering questions that involve conjunctive (i.e., the logical-AND operator) relationships. conditionals conjunctive/disjunc

A study by Noveck, Chierchia, and Sylvestre[1018] (performed using French) found that the exclusive interpretation occurs when a conclusion $QorR$ is more informative or relevant and is prompted by an implication (e.g., *but not both*).

Usage

Usage information is given elsewhere. 1248 logical-AND-expression syntax

Constraints

1257 Each of the operands shall have scalar type.

Commentary

The discussions in the various subsections of the logical-AND operator are applicable here. 1249 && operand type

C++

The operands are both implicitly converted to **`bool`** *(clause 4).* 5.15p1

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Semantics

1258 The `||` operator shall yield 1 if either of its operands compare unequal to 0; || operand compared against 0

Commentary

The discussion on the logical-AND operator is applicable here. 1250 && operand compare against 0

C++

5.15p1 *It returns* **true** *if either of its operands is* **true**, *and* **false** *otherwise.*

The difference in operand types is not applicable because C++ defines equality to return **true** or **false**. The difference in return value will not cause different behavior because **false** and **true** will be converted to 0 and 1 when required.

---

otherwise, it yields 0. 1259

Commentary

That is, a value of zero having type **int**.

|| result type

The result has type **int**. 1260

Commentary

&& result type 1252 The discussion in the various subsections of the logical-AND operator are applicable here.

C++

5.15p2 *The result is a* **bool**.

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; 1261

Commentary

&& evaluation order 1253 The discussion on the logical-AND operator is applicable here.

---

operator || sequence point

there is a sequence point after the evaluation of the first operand. 1262

Commentary

&& sequence point 1254 The discussion on the logical-AND operator is applicable here.

C++

5.15p2 *All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.*

&& sequence point 1254 The differences are discussed elsewhere.

If the first operand compares unequal to 0, the second operand is not evaluated. 1263

Commentary

&& second operand 1255 The discussion on the logical-AND operator is applicable here. An alternative way of looking at this operator is that `x || y` is equivalent to `x ? 1 : (y?1:0)`.

6.5.15 Conditional operator

1264

conditional-expression syntax

conditional-expression:
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

Commentary

The conditional operator is not necessary in that it is possible to implement the functionality it provides using other operators, statements, and sometimes preprocessing directives. However, C source code is sometimes automatically generated and C includes a preprocessor. The automated generation C source code is sometimes easier if conditional tests can be performed within an expression.

The second operand of a *conditional-expression* is *expression* which means that the **?** and **:** tokens effectively act as brackets for the second expression; no parentheses are required. For instance, `a ? (b , c) : d , e` can also be written as `a ? b , c : d , e`. The conditional operator associates to the right.

955 operator associativity

C++

5.16

conditional-expression: *logical-or-expression* *logical-or-expression* **?** *expression* **:** *assignment-expression*

By supporting an *assignment-expression* as the third operand, C++ enables the use of a *throw-expression*; for instance:

1288 assignment-expression syntax

```
1  z = can_I_deal_with_this() ? 42 : throw X;
```

Source developed using a C++ translator may contain uses of the conditional operator that are a constraint violation if processed by a C translator. For instance, the expression `x?a:b=c` will need to be rewritten as `x?a:(b=c)`.

Other Languages

In some languages (e.g., Algol 68) statements can return values (they are treated as expressions). For such languages the functionality of a conditional expression is provided by using an **if** statement (within an expression context). BCPL supports conditional expressions, using the syntax: *expression* `->` *expression* `,` *expression*.

Common Implementations

MetaWare High C (in nonstandard mode), some versions of `pcc`, and `gcc` support the syntax:

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

`gcc` allows the second operand to be omitted. The expression `x ? : y` is treated as equivalent to `x ? x : y`; `gcc` also supports compound expressions, which allow **if** statements to appear within an expression.

1313 compound expression

Coding Guidelines

Most developers do not have sufficient experience with the conditional operator to be familiar with its precedence level relative to other operators. Using parentheses removes the possibility of developers mistakenly using the incorrect precedence.

943.1 expression shall be parenthesized

Table 1264.1: Common token pairs involving `?` or `:` (to prevent confusion with the `:` punctuation token the operator form is denoted by `?:`) (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
`)` `?`	0.4	44.7	`?` *string-literal*	20.1	1.5
identifier `?`	0.1	44.0	`?:` *integer-constant*	28.7	0.3
identifier `?:`	0.1	40.3	`?` *integer-constant*	20.2	0.2
integer-constant `?:`	0.3	23.1	`?` identifier	43.9	0.1
string-literal `?:`	1.5	20.2	`?:` identifier	35.9	0.1
`)` `?:`	0.1	11.6	`?:` `(`	7.2	0.1
integer-constant `?`	0.1	9.6	`?` `(`	6.2	0.1
`?:` *string-literal*	21.0	1.6			

Constraints

The first operand shall have scalar type. 1265

Commentary

if statement 1743 controlling expression scalar type

This is the same requirement as that given for a controlling expression in a selection statement and is specified for the same reasons.

C++

5.16p1 *The first expression is implicitly converted to* ***bool*** *(clause 4).*

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Coding Guidelines

if statement 1743 controlling expression scalar type

Many of the issues that apply to the controlling expression of an **`if`** statement are applicable here.

conditional operator second and third operands

One of the following shall hold for the second and third operands: 1266

Commentary

conditional operator 1277 result

The result of the conditional operator is one of its operands. The following list of constraints ensures that the value of both operands can be operated on in the same way by subsequent operators (occurring in the evaluation of an expression).

Table 1266.1: Occurrence of the ternary `:` operator (denoted by the character sequence `?:`) having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
ptr-to	`?:`	ptr-to	29.5	**`int`**	`?:`	**`_int`**	5.7
other-types	`?:`	other-types	12.1	**`_char`**	`?:`	**`_char`**	3.4
`_int`	`?:`	**`_int`**	10.4	**`unsigned int`**	`?:`	**`unsigned int`**	2.2
`int`	`?:`	**`int`**	10.0	**`unsigned short`**	`?:`	**`unsigned short`**	1.2
`void`	`?:`	**`void`**	9.4	**`signed int`**	`?:`	**`_int`**	1.1
`unsigned long`	`?:`	**`unsigned long`**	7.9	**`char`**	`?:`	**`void`**	1.1
`_int`	`?:`	**`int`**	6.0				

— both operands have arithmetic type; 1267

Commentary

In this case it is possible to perform the usual arithmetic conversions to bring them to a common type.

Coding Guidelines

The guideline recommendation dealing with objects being used in a single role implies that the second and third operands of a conditional operator should have the same role.

1352.1 object used in a single role

1268 — both operands have the same structure or union type;

conditional operator structure/union type

Commentary

While the structure or union types may be the same, the evaluation of the expressions denoting the two operands may be completely different. However, a translator still has to ensure that the final result of both operands, used by any subsequent operators, is held in the same processor location.

1269 — both operands have void type;

Commentary

In this case the conditional operator appears either as the left operand of the comma operator, the top-level operator of an expression statement, or as the second or third operand of another conditional operator in these contexts. In either case alternative constructs are available. However, when dealing with macros that may involve nested macro invocations, not having to be concerned with replacements that result in operands having **void** type (invariably function calls) is a useful simplification.

1313 comma operator syntax

1270 — both operands are pointers to qualified or unqualified versions of compatible types;

conditional expression pointer to compatible types

Commentary

The discussion on the equality operators is applicable here.

1215 equality operators pointer to compatible types

C++

5.16p6

> *— The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9).*

These conversions will not convert a pointer to an enumerated type to a pointer to integer type.

If one pointed-to type is an enumerated type and the other pointed-to type is the compatible integer type. C permits such operands to occur in the same *conditional-expression*. C++ does not. See pointer subtraction for an example.

1159 subtraction pointer operands

1271 — one operand is a pointer and the other is a null pointer constant; or

Commentary

The discussion on the equality operators is applicable here.

1217 equality operators null pointer constant

1272 — one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**.

Commentary

The discussion on the equality operators is applicable here.

1216 equality operators pointer to incomplete type

C++

The C++ Standard does not support implicit conversions from pointer to **void** to pointers to other types (4.10p2). Therefore, this combination of operand types is not permitted.

```
int glob;
char *pc;
void *pv;

void f(void)
{
glob ? pc : pv; /* does not affect the conformance status of the program */
                // ill-formed
}
```

Semantics

The first operand is evaluated; 1273

Commentary

selection statement 1739 syntax

Much of the discussion on the evaluation of the controlling expression of an **if** statement is applicable here.

Common Implementations

controlling expression 1740 if statement

register 209 spilling

The only difference between evaluating this operand and a controlling expression is that in the former case it is possible for more processor registers to be in use, holding results from the evaluation of other subexpressions. (This likelihood of increased register pressure may result in spilling for processors with relatively few registers— e.g., Intel x86.) However, whether the overall affect of using a conditional operator is likely to be small, compared to an **if** statement, will depend on the characteristics of the expression and optimizations performed.

---

conditional operator sequence point

there is a sequence point after its evaluation. 1274

Commentary

selection statement 1739 syntax

full expression 1712

&& 1254 sequence point

Unlike the controlling expression in a selection statement, this operand is not a full expression, so this specification of a sequence point is necessary to fully define the evaluation order. The discussion on the logical-AND operator is applicable here.

C++

5.16p1 *All side effects of the first expression except for destruction of temporaries (12.2) happen before the second or third expression is evaluated.*

function call 1025 sequence point

The possible difference in behavior is the same as for the function-call operator.

---

conditional operator operand only evaluated if

The second operand is evaluated only if the first compares unequal to 0; 1275

Commentary

Because the operand might not be evaluated, replacing a conditional operator by a function call, taking three arguments, would not have the same semantics.

Coding Guidelines

&& 1255 second operand

The guideline discussion on the logical-AND operator is applicable here.

---

the third operand is evaluated only if the first compares equal to 0; 1276

Commentary

Either the second or the third operand is always evaluated, but never both.

1277 the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.[93)]

conditional operator result

Commentary

The conversion, to a common compatible type, ensures that the result of evaluating either operand has the same value representation.

Common Implementations

This places a requirement on the generated machine code to ensure that the result value is always left in the same register or storage location. Subsequent machine instructions will reference this single location.

Coding Guidelines

The difference between an **if** statement and a conditional operator is that in the latter case there is a result that can appear as the operand of another operator. Measurements of existing source suggest that developers have significantly more experience dealing with **if** statements than conditional operators. Using a conditional operator in a context in the visible source, where its result is not used (i.e., is not an operand to another operator), fails to take advantage of a developer's greater experience in dealing with **if** statements. However, such usage is rare and a guideline recommending the use of an **if** statement in this case is not considered worthwhile.

The conditional operator is not necessary for the writing of any program; it is always possible to rewrite source so that it does not contain any conditional operators. Similarly, source can be rewritten, replacing some **if** statements by conditional operators. How does the cost/benefit of using an **if** statement compare to that of using a conditional operator? The two main issues are comprehension and maintenance:

- From a reader's perspective, the comprehension of a conditional operator may, or may not, require more cognitive effort than comprehending an **if** statement. For instance, in the following example the use of an **if** statement highlights the controlling expression while the single assignment highlights that the array `a` is being modified (which needs to be deduced by reading two statements in the former case);

```
1  if (x)
2     a[y]=0;
3  else
4     a[z]=0;
5
6  a[x ? y : z]=0;
```

  When the expression being assigned to is complex, significant reader effort may be required to deduce that the two complex expressions are the same. Use of a conditional operator removes the need to make this comparison:

```
1  if (x)
2     a[complicated_expr]=y;
3  else
4     a[complicated_expr]=z;
5
6  a[complicated_expr]=(x ? y : z);
```

  When an expression in a substatement, of an **if** statement, contains a reference to an object that also occurs in the controlling expression, for instance:

1740 controlling expression if statement

```
1  if (x < 5)
2     a=10-x;
```

```
3   else
4      a=x;
5
6   a=(x < 5 ? 10-x : x);
```

readers may need to keep information about the condition in their minds when comprehending the source in both cases (i.e., maximum cognitive load is very similar, if not the same). When an expression in a substatement does not contain such a reference, readers do not need to evaluate information about the conditional to comprehend that statement. Compared to selection statements the maximum cognitive effort is likely to be less.

selection statement 1739 syntax

- In the following example a source modification requires that both `y` and `z` be assigned to `b`, instead of `a`, can be performed by editing one identifier in the case of the conditional operator usage, while the **`if`** statement usage requires two edits. However, a modification that required additional statements to be executed, if `x` were true (or false), would require far less editing for the **`if`** statement usage.

```
1   if (x)
2      a=y;
3   else
4      a=z;
5
6   a = (x ? y : z);
```

`if` statements occur in the visible source much more frequently than conditional operators. One reason is that in many cases the **`else`** arm is not applicable (only 18% of **`if`** statements in the visible form of the `.c` files contain an **`else`** arm). Because on this usage pattern readers receive more practice with the use of **`if`** statements. Given this difference in familiarity, it is not surprising that conditional operators are used less frequently than might be expected.

---

conditional operator attempt to modify

If an attempt is made to modify the result of a conditional operator or to access it after the next sequence point, the behavior is undefined. 1278

Commentary

function result 1007 attempt to modify

The discussion on the function-call operator result and its implementation details are applicable here.

C90

Wording to explicitly specify this undefined behavior is new in the C99 Standard.

C++

lvalue 721

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

Coding Guidelines

The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here.

Example

```
1   typedef struct {
2                   int mem_1;
3                  } A_S;
4   extern int glob;
5   extern A_S s_1, s_2;
6
7   void f(void)
8   {
9   int *p_l = &(((glob == 1) ? s_1 : s_2).mem_1);
10
11  *p_l = 2; /* Undefined behavior. */
12  }
```

1279 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result.

conditional operator arithmetic result

Commentary

There is no interaction between the types of the second and third operands and the first operand.

Other Languages

In Java:[508]

> *If one of the operands is of type T where T is* **byte**, **short**, *or* **char**, *and the other operand is a constant expression of type* **int** *whose value is representable in type T, then the type of the conditional expression is T.* 15.25

Coding Guidelines

If developers use the analogy of an **if** statement when reasoning about the conditional operator, they are unlikely to consider the effects of applying the usual arithmetic conversions to the operands (although your author is not aware of a case where a failure to take account of these conversions has resulted in a program fault). However, this may be due to use of the conditional operator being comparatively rare (and these guideline recommendations are not intended to cover rarely occurring construct).

1743 if statement controlling expression scalar type

0 guideline recommendations selecting

1234 role operand matching

For the purposes of these guideline recommendations, the role of the result is the same as the role of the second and third operands.

1280 If both the operands have structure or union type, the result has that type.

Commentary

Both operands are required to have the same structure or union type.

1268 conditional operator structure/union type

Common Implementations

The generated machine code may depend on what operation is performed on the result of the conditional operator. For instance, a simple member access `(x ? s1:s2).m` may generate the same machine code as if `(x ? s1.m : s2.m)` had been written. For more complicated cases, an implementation may load the address of the two operands into a register for subsequent indirect accesses.

Coding Guidelines

The form `(x ? s1:s2).m` has the advantage, over `(x ? s1.m : s2.m)`, that a change to the member selected only requires a single source modification (one of the two modifications needed in the latter form may be overlooked). Also the former form requires less effort to recognize as always accessing member `m`. (In the latter form the names of the two selected members needs to be checked.)

1281 If both operands have void type, the result has void type.

Commentary

This is one of the three cases where an operator does not return a value.

Coding Guidelines

If the operands have **void** type, the only affect on the output of the program is through the side effects of their evaluation. Such usage embedded in the visible form of an expression (that is not automatically generated, or the result of nested macro expansions) may be making assumptions about the order in which the operands of an expression are evaluated. This issue is covered by a guideline recommendation.

187.1 sequence points all orderings give same value

1282 If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types pointed-to by both operands.

conditional operator pointer to qualified types

Commentary

Specifying that the result has all the qualifiers of both operands requires that a translator make worst-case assumptions about subsequent operations. Type qualifiers on a pointed-to type do not affect the representation or alignment requirements of the pointer type, which means subsequent operators are able to treat either operand in the same way.

pointer 559 to qualified/unqualified types

In subsequent C sentences, in this C Standard paragraph, the term *appropriately qualified* refers to *all the type qualifiers* specified here (as is shown by the EXAMPLE).

EXAMPLE 1287 ?: common pointer type

Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; 1283

Commentary

The specification for forming composite types lists a set of properties the result must have. The previous C sentence ensured that the resulting pointed-to type has all the qualifiers of the two operands (two types that are pointers to differently qualified compatible types are not compatible). The composite type will contain at least as much, if not more, information than either of the types separately and will therefore be as restrictive (with regard to type checking).

composite type 642

C90

Furthermore, if both operands are pointers to compatible types or differently qualified versions of a compatible type, the result has the composite type;

The C90 wording did not specify that the appropriate qualifiers were added after forming the composite type. In:

```
1  extern int glob;
2  const enum {E1, E2} *p_ce;
3  volatile int *p_vi;
4
5  void f(void)
6  {
7  glob = *((p_e != p_i) ? p_vi : p_ce);
8  }
```

the pointed-to type, which is the composite type of the **enum** and **int** types, is also qualified with **const** and **volatile**.

---

if one operand is a null pointer constant, the result has the type of the other operand; 1284

Commentary

This places a requirement on the implementation to implicitly convert the operand that is a null pointer constant to the type of the other operand (different pointer types may use different representations for the null pointer). The discussion on the equality operators is applicable here.

null pointer 750 conversion yields null pointer
equality 1230 operators null pointer constant converted

---

otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**. 1285

Commentary

Although the null pointer constant may be represented by a value having a pointer to **void** type, the previous C sentence takes precedence. The discussion on the equality operators is applicable here.

null pointer 748 constant
equality 1231 operators pointer to void

C90

otherwise, one operand is a pointer to ***void*** *or a qualified version of* ***void****, in which case the other operand is converted to type pointer to* ***void****, and the result has that type.*

C90 did not add any qualifies to the pointer to **void** type. In the case of the **const** qualifier this difference would not have been noticeable (the resulting pointer type could not have been dereferenced without an explicit cast to modify the pointed-to object). In the case of the **volatile** qualifier this difference may result in values being accessed from registers in C90 while they will be accessed from storage in C99.

C++

The C++ Standard explicitly specifies the behavior for creating a composite pointer type (5.9p2) which is returned in this case.

Coding Guidelines

The coding guideline discussion on the equality operators is applicable here.

1216 equality operators pointer to incomplete type

Example

```
1  const int *p_ci;
2  volatile int *p_vi;
3
4  void f(void)
5  {
6  const volatile int *p_cvi = ((p_vi != (void *)0) ?
7                               p_ci :
8                               (volatile void *)0); /* Not a null pointer constant. */
9  }
```

1286 93) A conditional expression does not yield an lvalue.

footnote 93

Commentary

This footnote points out a consequence of specifications appearing elsewhere in the standard.

725 lvalue converted to value
729 array converted to pointer

```
1  struct {
2          int m1[2];
3         } x, y;
4  int glob;
5
6  void f(void)
7  {
8  (glob ? x : y).m1; /*
9                      * An array not a pointer to the first element.  Converting the
10                     * array to a pointer to its first element requires an lvalue.
11                     */
12 }
```

C++

If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue. 5.16p4

5.16p5

Otherwise, the result is an rvalue.

Source developed using a C++ translator may contain instances where the result of the conditional operator appears in an rvalue context, which will cause a constraint violation if processed by a C translator.

```
1  extern int glob;
2
3  void f(void)
4  {
5  short loc_s;
6  int loc_i;
7
8  ((glob < 2) ? loc_i : glob) = 3;  /* constraint violation */
9                                    // conforming
10 ((glob > 2) ? loc_i : loc_s) = 3; // ill-formed
11 }
```

Common Implementations

Some implementations (e.g., `gcc`) support, as an extension, a conditional operator yielding an lvalue.

Example

Using indirection, it is possible to assign to objects appearing as operands of a conditional operator.

```
1  (x ?  y :  z) = 0;     /* Constraint violation. */
2  *(x ? &y : &z) = 0;    /* Strictly conforming.  */
```

EXAMPLE ?: common pointer type

EXAMPLE The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types. 1287

Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

```
c_vp      c_ip     const void *
v_ip      0        volatile int *
c_ip      v_ip     const volatile int *
vp        c_cp     const void *
ip        c_ip     const int *
vp        ip       void *
```

Commentary

The effect is to maximize the number of type qualifiers and minimize the amount of type representation information (i.e., pointer to **`void`** is a generic pointer type) in the common type.

C90

This example is new in C99.

6.5.16 Assignment operators

1288

assignment-expression syntax

```
assignment-expression:

                conditional-expression
                unary-expression assignment-operator assignment-expression
assignment-operator: one of
                =  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
```

Commentary

The syntax is written so that these operators associate to the right. This means that in `x=y=z`, `z` is assigned to `y`, which is then assigned to `x`. Defining assignment as an operator is consistent with the specification that it returns a value. Its definition as an operator makes it possible for a full expression to contain more than one assignment.

955 operator associativity

1291 assignment value of

One benefit (to translator vendors) of compound assignment operators is that they remove the perceived need for translators to perform certain kinds of optimization. One potential disadvantage to developers of compound assignment operators is that they are likely to have the effect of causing vendors to write translators that do not search for certain kinds of optimization. The underlying reason is the same in both cases. The use of these operators affects the characteristics of the source that translator vendors expect to frequently encounter (e.g., because developers are expected to write `x*=3` rather than `x=x*3`). Thus vendors don't tune optimizers to search for assignments in code that don't make use of compound assignment operators (if this usage is possible).

0 source code characteristics

```
1  struct T {
2             struct fred *next;
3             /* ... */
4            } s_ptr;
5
6  s_ptr = s_ptr->next;
7  s_ptr ->= next;      /* There is no ->= operator. */
```

C++

```
assignment-expression:  conditional-expression logical-or-expression
assignment-operator assignment-expression throw-expression
```

5.17

For some types, a cast is an lvalue in C++.

1131 footnote 85

Other Languages

Most languages do not treat assignment as an operator, but as part of the syntax of the assignment statement. Languages in the Algol family use `:=` as the assignment token. The token `/=` denotes the not equal operator in some languages (e.g., Ada). Fortran contains the **`ASSIGN`** statement, which can be used to assign a label to an object having type **`INTEGER`**. Perl (and BCPL) support assigning multiple values to multiple objects within the same assignment (e.g., `v1, v2, v3 = e1, e2, e3`). Algol 68 does not treat assignment as an operator, but does treat compound assignment as operators. This results in `x +:= y := p -:= q := r` being parsed as `(x +:= y) := (p -:= q) := r`.

Common Implementations

base document 1

The base document supported `=+`, `=*`, and the reversed form of the other compound assignment operators.[723]

Coding Guidelines

What are the costs and benefits of using multiple assignment operators in the same full expression?

The potential benefits, compared to source code where full expressions contain at most one assignment operator, include:

- The resulting program image may contain higher-quality (faster and/or smaller) machine code. However, this benefit may not exist (if the translator used is sufficiently sophisticated that it generates comparable machine code for both cases). The benefit may also be of no significance— the difference in program performance is not noticeably different and the size savings is of no consequence.
- When carefully reading the source (rather than scanning it quickly), a multiple assignment may require less cognitive effort to comprehend than two assignments. For instance, if the objects being assigned to are denoted by some complicated expression, it is necessary to deduce that, for instance, `complicated_y` represents the same object in both cases. This deduction does not need to be performed when a multiple assignment is used:

```
1  complicated_x = complicated_y = some_expression;
2
3  complicated_y = some_expression;
4  complicated_x = complicated_y;
```

- Use of the form `x = y = exp` is not usually taken to imply a causal connection between `x` and `y`, while an assignment of the form `x=y` often implies a causal connection between the two objects. When `exp` is complicated, a multiple assignment may help clarify that any causal association should be to it, not between the two objects assigned to.

The main potential costs are reader miscomprehension of the expression and a possible increase in the maximum cognitive load required to comprehend the expression, as follows.

reading kinds of 770

- The miscomprehension can occur because of the techniques developers use to read source code. When searching for objects that are modified, developers often visually scan along the left edge of the source (making the assumption that all identifiers denoting modified objects appear along this edge). If multiple assignment operators occur within the same expression, it is possible that only the first one of them will be seen.

postfix ++ result 1047

- Increase in cognitive load can be caused by the task switch needed to update a reader's mental model of a program's state. This issue is discussed elsewhere.

Multiple objects are sometimes initialized to the same value in a single statement (e.g., `a=b=c=d=0;`). The developer has saved the typing of a few characters at the expense of less-readable source code and an increased likelihood that changes to the source lead to errors. For instance, assigning a different value to `c` implies that the original assignment to `c` would be removed; poor editing could create the expression `a=b==d=0;` (failure to spot the original assignment to `c`, resulting in it being left in the source, could introduce errors if it occurred after a different value had been stored into `c`).

Cg 1288.1

A full expression shall contain at most one assignment operator, and it shall occur as the *top-level* operator in that expression.

controlling expression if statement 1740

The issue of assignment operators occurring in other types of statements that contain expressions is discussed elsewhere.

Example

```
extern int ***x,
           *y;

void f(void)
{
**x+++=***x++;
*y*=*y;
*y-=***x---=*y--;
}
```

Usage

For a comparison with load frequencies see Table 976.2.

Table 1288.1: Common token pairs involving the assignment operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier **%=**	0.0	100.0	**v++ =**	7.6	0.7
identifier **/=**	0.0	99.3	**+=** *integer-constant*	21.7	0.3
identifier **>>=**	0.0	99.3	**|=** identifier	77.0	0.2
identifier **<<=**	0.0	97.5	**+=** identifier	68.0	0.2
identifier **+=**	0.3	96.3	**>>=** *integer-constant*	87.1	0.1
identifier ***=**	0.0	96.0	**-=** *integer-constant*	24.2	0.1
identifier **-=**	0.1	95.2	**&=** *integer-constant*	12.4	0.1
identifier **|=**	0.3	93.9	**|=** *integer-constant*	10.7	0.1
identifier **&=**	0.1	93.1	**-=** identifier	65.1	0.1
identifier **=**	9.4	90.9	**+= (**	6.5	0.1
identifier **^=**	0.0	85.9	**|= (**	12.0	0.1
&= ~	75.0	52.5	**<<=** *integer-constant*	85.1	0.0
= +v	0.0	45.1	**/=** *integer-constant*	52.1	0.0
= *floating-constant*	0.1	15.7	***=** *integer-constant*	39.8	0.0
= *character-constant*	0.8	14.2	**^=** *integer-constant*	34.5	0.0
= -v	1.6	12.0	**%=** *integer-constant*	31.5	0.0
] ^=	0.0	11.1	**&=** identifier	8.6	0.0
= &v	1.9	10.2	**%=** identifier	68.1	0.0
= *v	1.1	9.9	**^=** identifier	46.4	0.0
= *integer-constant*	19.6	9.0	***=** identifier	44.2	0.0
] =	21.8	6.8	**/=** identifier	34.6	0.0
= identifier	62.5	6.5	**<<=** identifier	13.4	0.0
= sizeof	0.3	5.9	**>>=** identifier	10.5	0.0
] &=	0.2	5.7	**#error =**	16.9	0.0
] |=	0.4	4.6	**-= (**	7.0	0.0
= (	9.1	3.5	**/= (**	5.8	0.0
***=** *floating-constant*	6.3	1.6	**^= (**	13.9	0.0

Table 1288.2: Occurrence of executed store instructions (as a percentage of all instructions executed) in two different kinds of functions (*Leaf* functions do not call any other functions, while *Non-Leaf* do). Adapted from Calder, Grunwald, and Zorn.[192]

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	34.3	7.7	eqntott	0.0	11.4
ditroff	8.3	8.3	espresso	6.5	3.9
tex	15.1	9.8	gcc	9.6	12.0
xfig	8.0	11.7	li	0.0	16.3
xtex	8.3	11.2	sc	1.2	11.1
compress	83.5	9.2	Mean	15.9	10.2

Constraints

assignment operator modifiable lvalue

An assignment operator shall have a modifiable lvalue as its left operand. 1289

Commentary

modifiable lvalue 724

An object declared using the **`const`** is an lvalue, but it is not modifiable.

C90

lvalue 721

The C99 Standard has removed the requirement, that was in C90, which lvalues refer to objects. This has resulted in the conformance status of the assignment `1=3` changing from a constraint violation to undefined behavior. The lvalue `1` does not designate an object and is not const-qualified. Therefore it is not ruled out from being modifiable in C99.

lvalue 721

Common Implementations

Some implementations support an extension that allows the result of the cast operator to be an lvalue (e.g., assignments of the form `(char)i = 3` are supported).

Example

In some cases an implementation is likely to diagnose a syntax violation, in an assignment expression, rather than this constraint violation.

```
1    (int)x = 0;   /* Syntax violation. */
2   +(int)x = 0;   /* Syntax is valid, but violates this constraint. */
```

Semantics

An assignment operator stores a value in the object designated by the left operand. 1290

Commentary

modified objects 192
received correct value
assignment 1293
when side effect occurs

The standard does not specify when the value is stored, only that it occurs between the previous and next sequence point. Neither does the standard specifies how the store should be implemented. For instance, storing into an object having a floating type does not require that the value be treated as having a floating type; simple assignment of `a` to `b` may be implemented using machine code that copies blocks of bits, or as the instruction sequence load floating value/store floating value. For floating types this difference can matter because assigning a signaling NaN will not raise an exception in the former case, but will in the latter.

Other Languages

Assignment is not universal to all programming languages. Some pure functional languages regard assignment as causing a side effect that make it difficult to mathematically prove the correctness of source code. They do not contain any means of assigning a value to an object.

Some languages, often used in distributed computing, perform what is sometimes known as *deep assignment*. For instance, if `X` refers to a tree-like data structure, then assigning `X` to `Y` has the effect of making a copy of the tree and assigning a reference to it to `Y`; `X` and `Y` will then point at different trees, which contain nodes having the same values (apart from the pointers to other nodes).

Common Implementations

Whether the store into an object specified in the source code actually occurs in the generated machine code is invisible to the developer. Optimizers frequently try to remove stores by keeping the value in a register (perhaps until a new value is assigned). For processors with many registers, functions containing a small number of automatic objects may never need to store assignments to those objects.

Usage

A study by Lepak, Bell, and Lipasti[835] investigated value locality with respect to store operations (using the SPEC95 benchmarks). They defined a *silent store* to be a store operation that does not change the system state (i.e., the value being written matches the value already held at the location being stored to). They also defined *program structure store value locality* (PSSVL) to refer to the same value being stored from the same program location and *message-passing store value locality* (MPSVL) to refer to the same value being stored to the same address in storage (which may be holding different objects at different times during the execution of a program).

976 value locality

Table 1290.1: Percentage of stores that are *silent*. The results from two instruction sets, the PowerPC (PPC) and SimpleScalar (SS), are given for silent stores. The measurements for Program Structure Store Value Locality (PSSVL) and Message-Passing Store Value Locality (MPSVL) are for the PowerPC only. Adapted from Lepak, Bell, and Lipasti.[835]

Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)	Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)
go	38/27	30	36	tomcatv	47/33	40	45
m88ksim	68/62	56	65	swim	34/26	20	19
gcc	53/46	37	49	mgrid	23/ 7	24	17
compress	42/39	35	16	applu	37/35	35	28
li	34/20	32	34	apsi	21/25	22	20
ijpeg	43/33	52	46	fpppp	15/15	15	14
perl	49/36	39	42	wave5	25/22	30	20
vortex	64/55	71	57				

Results for two processors and their associated translators are given in Table 1290.1. Differences in internal housekeeping operations, such as saving registers as part of a function-call operation, can affect the results. The same authors[110] found that zero was the most common silent store value (46% of silent stores), with one being 7%, 2 to 9 being 4%, and values greater than 100 million being 26%. For floating-point, values greater than 100 million were the most common at 65%, with zero being 37%.

1004 function call preparing for

1291 An assignment expression has the value of the left operand after the assignment, but is not an lvalue.

assignment value of

Commentary

Using the value of an assignment expression does not mean that the side effect of storing that value in the left operand has taken place.

If a guarantee is needed that the assignment will have taken place before the value is used, or an lvalue is required. The expression `((a=b), a)` *can be used.*

WG14/N847

Two interpretations have been put on this wording:

- *the value of the assignment expression is the value that will also be stored in the left operand ("same-value" semantics);*
- *the value of the assignment expression is the result of reading the left operand after storing the value in it ("write-then-read" semantics).*

These two have different results when the left operand is a volatile object that can be changed by external causes (such as a clock or a memory-mapped device register). This ambiguity needs to be resolved.

Consider the code:

```
int x;
extern volatile int system_timer;  // precision of 1 microsecond
extern volatile int serial_port;   // writing sends a word, reading
                                   //   returns the next word received
// ...
x = system_timer = 42;  // statement 1
serial_port = 66;       // statement 2
```

With same-value semantics, statement 1 will set x to 42 and will send the value 66 to the serial port. With write-then-read semantics, statement 1 will set x to some other value (the change in the timer between writing to it and reading it back).

*More important, though, is the effects of statement 2 in write-then-read semantics. Because a statement expression is evaluated for its side effects, it is reasonable to require the value of the assignment statement to be determined before being thrown away (in particular, there is *no* statement in the Standard as to when the value of the assignment expression is or is not evaluated). This means that statement 2 always has the side effect of reading a word from the serial port, and there is no way to write without reading.*

C++

5.17p1 *. . . ; the result is an lvalue.*

The C++ DR #222 (which at the time of this writing is at the drafting stage) queries some of the consequences of the result being an lvalue.

Source developed using a C++ translator may contain assignments that are a constraint violation if processed by a C translator.

```
1  extern int glob;
2
3  void f(void)
4  {
5  int x;
6  volatile int y;
7
8  (glob += 5) += 6; /* constraint violation */
9                    // current status undefined behavior, object modified
10                   // twice between sequence points.  The response to DR #222
11                   // may add a sequence point, making the behavior defined
12
13 x = y = 0; /* equivalent to y=0; x=0; */
14            // equivalent to y=0; x=y;
15 }
```

Other Languages

In the past languages have not generally treated assignment as an operator. Whether more recent languages are treating it as an operator because of the effects of the widespread teaching of C to students over several decades or for other reasons is not known.

Coding Guidelines

Experience shows that developers sometimes incorrectly assume that the result of the assignment operator is the value of the right operand before it is converted to the type of the left operand.

Rev 1291.1 If the result of an assignment operator is used and the types of its two operands are different, the source shall be checked to ensure that the value of the right operand is not being expected.

Example

In the first statement in the following the value of LONG_MAX will be implicitly converted (to the value 2147483647) and assigned to ui.

```
#include <limits.h>

void f(void)
{
unsigned char uc;  /* Assume 8 bits.  */
unsigned short us; /* Assume 16 bits. */
unsigned int ui;   /* Assume 32 bits. */

uc=us=ui=LONG_MAX;
ui=us=uc=LONG_MAX;
}
```

This value is what is assigned to us, but first it has to be converted, giving a value of 65535. This value is then assigned to uc, but first it must be converted, giving a value of 255. In the second statement the value 255 is assigned to all three objects.

1292 The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand.

assignment result type

Commentary

This requirement is consistent with the value of an assignment expression being that of the left operand.

Other Languages

Algol 68 treats an assignment as returning an lvalue, so it is possible to write:

```
INT x;
REF INT r;
r := x := 42;
```

which causes r to point to x, which is assigned the value 42.

Coding Guidelines

The incorrect developer assumption described in the previous C sentence also applies to the type of the result.

Rev 1292.1 If the result of an assignment operator is used and the types of its two operands are different, the source shall be checked to ensure that the type of the right operand is not being expected.

Example

```
extern unsigned char uc;
extern float fl;

void f(void)
{
if ((fl += 1) == 3)           /* Equality comparison of floating type. */
   ;
if (((uc = 3.5) + 0.5) == fl) /* Adding 0.5 to an unsigned char?       */
   ;
}
```

assignment when side effect occurs

The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point. 1293

Commentary

This is a requirement on the implementation. Like other operators that generate side effects, the exact time when the side effect occurs is not specified, only the bounds between when it must occur. Most assignment operators occur in the context of an expression statement, which is a full expression and hence has a sequence point after its evaluation.

sequence 187 points expression 1731 statement syntax full expression 1715 expression statement full expression 1720 sequence point

C++

The C++ Standard does not explicitly state this requirement.

Other Languages

In languages that do not treat assignment as an operator, updating the stored value is the last operation performed in that statement (which may contain function calls that modify objects).

Coding Guidelines

sequence 187 points sequence 187.1 points all orderings give same value

It is not always possible to predict the ordering of sequence points chosen by a translator, and the guideline recommendation dealing with expression order evaluation is applicable here.

---

assignment operand evaluation order

The order of evaluation of the operands is unspecified. 1294

Commentary

Many developers have a mental model of assignment that involves evaluating one of the operands first. (This is often a hang over from how they were first taught to write programs.) The evaluation of the operands has the same behavior as most other binary operators in C. This sentence is simply a restatement of this fact.

expression 944 order of evaluation

C++

The C++ Standard does not explicitly make this observation.

Other Languages

Few languages specify an ordering on the evaluation of the left and right operands of an assignment, even if they do not consider assignment to be an operator. Java specifies a left-to-right evaluation order.

Common Implementations

Unsophisticated translators will evaluate the right operand first, on the basis that it is likely to be the most complex (and therefore require the greater number of temporary registers). More sophisticated translators will estimate the complexity of both operands and evaluate the most complex one first.

Coding Guidelines

The issues generated by some developers, who have a mental model of assignment that involves one operand always being evaluated before the other, are the same as for the other binary operators. The guideline recommendation dealing with expression order evaluation is applicable here.

expression 944 order of evaluation sequence 187.1 points all orderings give same value

Example

```
1  #include <stdio.h>
2
3  char arr[20];
4
5  void f(void)
6  {
7  arr[printf("Hello ")] = printf("World\n");
8  }
```

1295 If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

Commentary

In some cases it is possible to create an lvalue from the result returned by the assignment operator (these invariably involve constructs whose implementation involves the use of temporary storage locations). The order of evaluation of most operands is unspecified and when it is specified, a sequence point also occurs. This C sentence effectively specifies that there is no requirement for implementations to support developer access to these temporary objects.

C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue. 721 lvalue

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

Common Implementations

The temporary storage locations used internally by an implementation are usually allocated within the stack frame of the function invocation that uses them. As such, they will cease to exist when that function returns. These storage locations may also be used to hold temporary results from the evaluation of other operations.

Coding Guidelines

For this undefined behavior to occur an expression must contain more than one assignment operator. The guideline recommendation dealing with the use of multiple assignment operators in the same expression is therefore applicable. The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here. 1288.1 full expression at most one assignment

Example

```
struct {
        int m[10];
       } x, y;

void f(void)
{
unsigned char uc_1;
int *p_1 = &((x = y).m[3]);

*p_1 = 4;
p_1 = &((x = y).m[uc_1=0, 2]);

/*
 * Result accessed before sequence point, but behavior is unspecified(?)
 * because the object written to is also read from in the same subexpression.
 */
y.m[4] = (*p_1 = &((x = y).m[3])) + *(p_1++);
}
```

6.5.16.1 Simple assignment

Constraints

1296 One of the following shall hold:[94)] simple assignment constraints

Commentary

equality 1213 operators constraints

This list is very similar to that given for the equality operators.

C++

The C++ Standard does not provide a list of constraints on the operands of any assignment operator (5.17). Clause 12.8 contains the specification that leads the following difference:

C1.8 *The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:*

```
struct X { int i; };
struct X x1, x2;
volatile struct X x3 = {0};
x1 = x3;     // invalid C++
x2 = x3;     // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

— the left operand has qualified or unqualified arithmetic type and the right has arithmetic type; 1297

Commentary

operators 702 cause conversions modifiable lvalue 724 assignment 1289 operator modifiable lvalue

Any arithmetic value can be assigned to any object that has an arithmetic type. Conversions are specified to handle all cases where the two arithmetic types are not the same. This constraint does not prohibit the type of the left operand being const-qualified. However, such an object would not be a modifiable lvalue, and such usage would violate a constraint that applies to all assignment operators.

C++

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

The conversions in clause 4 do not implicitly convert enumerated types to integer types and vice versa.

```
1  extern int glob;
2
3  enum {E1, E2};
4
5  void f(void)
6  {
7  glob = E1; /* does not affect the conformance status of the program */
8             // ill-formed
9  }
```

Other Languages

Strongly typed languages sometimes require the names of the types used to declare the operands to be the same. There are usually special rules to cover literal values.

Common Implementations

Simple assignments of the form `x=y;` are very common. Some processors (e.g., the Motorola 68000[968]) support memory-to-memory copies (no implicit conversions are performed, so both operands must have the same representation). A processor that supports memory-to-memory copies needs to encode two addresses—the source and destination— within the instruction. Such encoding requires a large number of bits (some Motorola 68000 instructions are 13 bytes long). Considering the design of instruction encodings from a more global perspective, there are cost/benefit processor implementation advantages in having fixed-width instructions, and there is often a greater benefit to be had in using the available instruction bits to specify other operations. For these reasons it is very rare to find a modern processor that supports memory to memory operations. On most processors the assignment `x=y` is implemented as "load value into register, store register contents into object". Depending on the local source code context, optimizers may be able to reuse the value held in the register.

Coding Guidelines

The guideline recommendation for binary operators with an operand having an enumeration type is applicable here.

517.3 enumeration constant as operand

1298 — the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right;

assignment structure types

Commentary

Structure/union assignment was introduced in C90; it was not supported in the base document. Structure assignment was first specified in a document listing extensions made to the base document.

1 base document

Structure assignment can be more efficient than using the `memcpy` library function. This is because the implementation of `memcpy` has to assume worst-case alignment and copy a byte at a time (some very good implementations do better). The translator knows the alignment of an object having structure type and may be able to copy its contents in larger chunks. It may even be worthwhile to copy an element at a time.

601 footnote 42

Structures sometimes have an array as their only member, which effectively allows arrays to be assigned.

C++

Clause 13.5.3 deals with this subject, but does not discuss this particular issue.

Other Languages

It seems to be quite common for the first versions of a language definition to not support structure assignment. Such support is often added in later revisions of language definitions.

Common Implementations

Copying a structure a member at a time (or even in larger units— e.g., a long's worth of bits) produces the fastest code, but at the potential cost of a large number of instructions. Generating a loop is often more compact (except for the case of small structures), but runs more slowly because of the housekeeping overhead of controlling a loop (and the backward jump, potentially flushing the instruction cache). Most implementations generate machine code to copy a member at a time for small structure types and looping machine code for larger one. Depending on the method selected by the implementation padding bits, may or may not also be copied.

0 cache

Coding Guidelines

Use of the assignment operator can result in the object assigned to containing a different sequence of bits than if the `memcpy` library function had been used. In the former case a call to the `memcmp` library function may not return zero, while in the latter it will always return zero. The reason for using `memcmp` is that the equality operators do not support operands having a structure or union type. A strong case can be made for using `memcmp` to compare two structure objects, other than the (usually) spurious efficiency reason; if new members are added to a structure type, a comparison using `memcmp` will not have to be modified, while one that checks individual members will need to be updated to reflect the presence of the new member.

1213 equality operators constraints

Cg 1298.1

Objects having structure or union type that are compared using the `memcmp` library function shall not appear as the immediate operands of an assignment operator.

Example

```
#include <stdio.h>
#include <string.h>

struct {
        long m1;
        char m2;
       } x, y;

int main(void)
{
x = y;
if (memcmp(&x, &y, sizeof(x)) == 0)
   printf("either there are no padding bits, the assignment operator copies all"
          "padding bits, or the padding bits happen to be the same\n");

memcpy(&x, &y, sizeof(x));
if (memcmp(&x, &y, sizeof(x)) != 0)
   printf("something wrong here\n");

memcpy(&x, &y, sizeof(x));
x.m1=y.m1;
if (memcmp(&x, &y, sizeof(x)) != 0)
   printf("member assignment affects padding bits\n");
}
```

---

pointer qualified/unqualified versions

— both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right; 1299

Commentary

Given the following declarations (where `Q_0` and `Q_1` represent zero or more qualifiers, and `T` is an unqualified type; with the lowercase names denoting the same entities):

```
T Q_1 * Q_0 left_operand;
t q_1 * q_0 right_operand;
```

compatible type 631 if assignment operator 1289 modifiable lvalue

this constraint requires that: the types `T` and `t` be compatible, and that all the qualifiers in `q_1` also be in `Q_1`. It is permitted for `Q_1` to contain qualifiers that are not in `q_1`. The case of `Q_0` containing the **`const`** qualifier is covered by wording given elsewhere.

C++

The C++ wording (5.17p3) requires that an implicit conversion exist.

pointer 746 converting qualified/unqualified

The C++ requirements (4.4) on which implicit, qualified conversions are permitted are those described in the Smith paper (discussed elsewhere).

The pointer assignments supported by C++ are a superset of those supported by C. Source developed using a C++ translator may contain constraint violations if processed by a C translator, because it contains assignments between incompatible pointer types. The following example illustrates differences between the usages supported by C and C++ when types using two levels of pointer are declared.

```
1  void Jon_Krom(void)
2  {
3  /*
4   * The issue of what is safe or unsafe is discussed elsewhere.
5   * An example of case 3 is given in the standard.
6   */
7
8  typedef int T; /* for any type T */
9
10 T       *       *  ppa  ;
11 T       *       *  ppb  ;
12
13 T       * const *  pcpa ;
14 T       * const *  pcpb ;
15
16 T const *       * cppa  ;
17 T const *       * cppb  ;
18
19 T const * const * cpcpa ;
20 T const * const * cpcpb ;
21
22                   //  Safe       Allowed     Allowed
23                   //   or         in          in
24                   // Unsafe       C99         C++
25                   // --------------------------------
26 ppb   = ppa ;     //  Safe        Yes         Yes      1
27 pcpb  = ppa ;     //  Safe        Yes         Yes      2
28 cppb  = ppa ;     //  Unsafe      No          No       3
29 cpcpb = ppa ;     //  Safe        No          Yes      4
30
31 ppb   = pcpa ;    //  Unsafe      No          No       5
32 pcpb  = pcpa ;    //  Safe        Yes         Yes      6
33 cppb  = pcpa ;    //  Unsafe      No          No       7
34 cpcpb = pcpa ;    //  Safe        No          Yes      8
35
36 ppb   = cppa ;    //  Unsafe      No          No       9
37 pcpb  = cppa ;    //  Unsafe      No          No      10
38 cppb  = cppa ;    //  Safe        Yes         Yes     11
39 cpcpb = cppa ;    //  Safe        Yes         Yes     12
40
41 ppb   = cpcpa ;   //  Unsafe      No          No      13
42 pcpb  = cpcpa ;   //  Unsafe      No          No      14
43 cppb  = cpcpa ;   //  Unsafe      No          No      15
44 cpcpb = cpcpa ;   //  Safe        Yes         Yes     16
45 }
```

Other Languages

Languages that support some form of type qualifier (e.g., **`readonly`**) usually have similar asymmetric constraints on pointer assignment.

Coding Guidelines

If differently qualified pointers are used to access the same object, in an overlapping time frame, it is possible that readers of the source will make an incorrect assumption about the type of one or more of them. For instance, if a pointer to **`const`** T and a pointer to T both point at the same object, a developer (who is not aware of this state of affairs) may assume that the pointer to const-qualified type cannot have its value changed. Qualified pointers are a special case of object aliasing and the conclusion reached there is applicable.

971 object aliased

1300 — one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **`void`**, and the type pointed to by the left has all the qualifiers of the type pointed to by

the right;

Commentary

Either operand can have a pointer to **void** type. This combination of operands provides implicit support for the concept of a generic pointer type (no explicit casts are required).

generic pointer 523

The issues relating to qualifiers are the same as those in the previous C sentence.

C++

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

The C++ Standard only supports an implicit conversion when the left operand has a pointer to **void** type, 4.10p2.

```
1  char *pc;
2  void *pv;
3
4  void f(void)
5  {
6  pc=pv; /* does not affect the conformance status of the program */
7         // ill-formed
8  }
```

Coding Guidelines

The coding guideline discussion on the equality operators is applicable here.

equality operators pointer to incomplete type 1216

— the left operand is a pointer and the right is a null pointer constant; 1301

Commentary

A null pointer constant can be converted to any pointer type.

null pointer conversion yields null pointer 750

Other Languages

Languages that support pointer data types invariably have some form of null pointer that can be assigned to an object having any other pointer type.

Coding Guidelines

The issue of explicit casts is discussed elsewhere.

pointer converted to pointer to void 744

---

or— the left operand has type **_Bool** and the right is a pointer. 1302

Commentary

This specification is consistent with operands in other contexts having pointer type being implicitly compared for equality with 0 (the null pointer constant).

|| operand compared against 0 1258
&& operand compare against 0 1250
if statement operand compare against 0 1744
_Bool converted to 680

C90

Support for the type **_Bool** is new in C99.

C++

Support for the type **_Bool** is new in C99 and is not specified in the C++ Standard. However, the C++ Standard does specify (4.12p1) that rvalues having pointer type can be converted to an rvalue of type **bool**.

Other Languages

Other languages do not usually perform an implicit comparison when the operand has point type.

Coding Guidelines

Support for the `_Bool` type is new in C99 and at the time of this writing there is insufficient experience available in its use to know if any guideline recommendation is worthwhile.

Semantics

1303 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

simple assignment

Commentary

This defines the term *simple assignment*. Because it is the most commonly seen form of assignment operator developers invariably shorten it to *assignment*. The less-frequent forms of assignment are known by longer terms (see Table 912.1).

The type of the result of the assignment operator is the unqualified type of its left operand. The standard does not guarantee that assignment is an atomic operation unless the left operand has type `sig_atomic_t`. Assigning an object having a structure type is not always identical to assigning its members individually. There may be an order dependency.

1292 assignment result type
192 modified objects received correct value

```
1  struct S_Pair;
2
3  typedef struct Object {
4                          struct S_Pair *addr;
5                          int tag;
6                         } Object;
7  struct S_Pair {
8                  Object car;
9                  Object cdr;
10                };
11
12 Object x;
13
14 void copy_obj(void)
15 {
16 x = x.addr->cdr;
17
18 /* is not the same as: */
19
20 x.addr = x.addr->cdr.addr;
21 x.tag  = x.addr->cdr.tag;
22 }
```

Common Implementations

In most implementations the store operation for an assignment operator, for scalar types whose bit representation is not wider than the width of the processor data bus (which is unlikely to include complex types), is an atomic operation. Unless objects having structure or union types that can fit in a single register, stores of their values are unlikely to be atomic operations.

A surprising number of assignment operations store a value that is equal to the value already held in memory. Researchers[278] are starting to adapt algorithms used to detect redundant load operations to optimize away such redundant store operations.

1288 assignment-expression syntax
190 redundant code

Coding Guidelines

If the right operand does not have the same type as the left operand, it will be implicitly converted to that type. Assignment is different from all other binary operators in that it can cause the value of one of its operands to be implicitly converted to a type that has a lower integer rank, or a narrower floating type. A consequence of such a conversion is that the assignment `x=y` does not guarantee that a subsequent equality operation `x==y` will be true. This situation does not occur if the guideline recommendation specifying the use of a single integer type is followed.

480.1 object int type only

operand 653 convert automatically

The guideline recommendations applicable to assignment include those given for conversions.

---

assignment value overlaps object

If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; 1304

Commentary

This requirement (on programs) is needed because implementations may choose to perform an assignment by copying values from the right operand to the left operand a byte at a time. It would be surprising if the assignment x=x did not always deliver the expected result (an unchanged x; unless it is declared with the **volatile** qualifier).

For a given implementation it is possible for two types to have the same size and to overlap exactly, but not be compatible (e.g., types **int** and **long**). The conformance status of a program cannot depend on the implementation, hence the additional requirement on compatible types. The issue of overlapping objects is primarily of importance when assigning members of an object having a union type.

```
union {
      int m_1;
      long m_2;
      int m_3;
      struct {
             char m_4;
             long m_5;
             } m_6;
      } x;

void f(void)
{
x.m_1=x.m_2;     /* Not compatible types. */
x.m_1=x.m_3;     /* Covered by this requirement. */
x.m_2=x.m_6.m_5; /* Overlap (if it exists) not exact. */
}
```

Requiring that all overlapping assignments work as expected would sometimes involve the use of temporary storage locations. Such an overhead during program execution was considered to be excessive. It would require the use of the `memmove` library function rather than the `memcpy` library function.

C++

object types 475

The C++ Standard requires (5.18p8) that the objects have the same type. Even though the rvalue may have the same type as the lvalue (perhaps through the use of an explicit cast), this requirement is worded in terms of the object type. The C++ Standard is silent on the issue of overlapping objects.

```
enum E {E1, E2};
union {
       int m_1;
       enum E m_2;
      } x;

void f(void)
{
x.m_1 = (int)x.m_2; /* does not change the conformance status of the program */
                    // not defined?
}
```

Other Languages

Some languages treat the value of the right operand as being independent of the object referred to by the left operand. In these languages if there is any possibility of overlap and the generated machine code has to

perform the assignment by copying multiple storage units of the value, then a temporary has to be used to hold the value so that partial stores into the object do not modify the value being assigned.

1305 otherwise, the behavior is undefined.

Commentary

Objects can overlap in a variety of ways and implementations can copy objects using a variety of techniques. The Committee chose not require implementations to support any particular set of behaviors.

Common Implementations

For objects having scalar types, the copying is likely to be performed by loading the value into a register. Once the value has been completely loaded, how the subsequent store is performed does not affect the final result. It is those assignments that require more than one load instruction (where the source and destination overlap) where the behavior can vary. The result will depend on the relative addresses of the objects and whether the copying starts at the lowest or the highest byte of the objects.

Coding Guidelines

No guideline recommendation is given for this situation because occurrences of this usage are assumed to be rare.

1306 EXAMPLE 1 In the program fragment

narrower example

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
        /* ... */
```

the **int** value returned by the function may be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable **c** should be declared as **int**.

Commentary

There are also less visibly obvious situations involving a value of -1. For instance, the EOF macro.

296 limit case labels

Coding Guidelines

The coding guideline discussion on the assignment operator appearing in a controlling expression is applicable here.

1740 controlling expression if statement

1307 94) The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" ~~which~~and thus removes any type qualifiers ~~from the type category of the expression~~ that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **intvolatile*const**).

footnote 94

Commentary

The definition of type category does not include qualifiers. There is even a C sentence that says so. All of the contexts where the conversion is not performed do not apply to the immediate left operand of the assignment operator.

553 type category
556 qualifiers representation and alignment
725 lvalue converted to value

The wording was changed by the response to DR #272.

C++

Even though the result of a C++ assignment operator is an lvalue, the right operand still needs to be converted to a value (except for reference types, but they are not in C) and the asymmetry also holds in C++.

EXAMPLE 2 In the fragment: 1308

```
char c;
int i;
long l;

l = (c = i);
```

the value of **i** is converted to the type of the assignment expression **c = i**, that is, **char** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **long int** type.

Commentary

assignment 1291 value of

While C++ supports the form `(l = c) = i;`, C does not.

Coding Guidelines

full expression 1288.1 at most one assignment

The coding guideline discussion on the use of more than one assignment operator in an expression is applicable here.

EXAMPLE const pointer

EXAMPLE 3 Consider the fragment: 1309

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;       // constraint violation
*cpp = &c;      // valid
*p = 0;         // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object **c**.

Commentary

The term *unsafe* is often used to refer to an operation, which although itself is conforming, whose resulting value may subsequently be a cause of undefined behavior.

6.5.16.2 Compound assignment

Constraints

compound assignment constraints

For the operators += and -= only, either the left operand shall be a pointer to an object type and the right shall have integer type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type. 1310

Commentary

addition 1154 operand types

The discussion on the constraints for the additive operators are applicable here.

Coding Guidelines

boolean role 476 symbolic name 822 addition 1154 operand types

The issue of operands having a boolean or symbolic role is discussed elsewhere.

Table 1310.1: Occurrence of assignment operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand, occurrences below 2.3% were counted as *other-types*). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
other-types	-=	other-types	34.5	float	/=	float	6.4
other-types	+=	other-types	33.5	unsigned short	|=	_int	6.2
other-types	=	other-types	32.8	ptr-to	+=	_int	6.2
int	%=	_int	31.0	unsigned long	|=	int	6.1
ptr-to	=	ptr-to	29.7	unsigned int	-=	unsigned int	5.9
int	*=	_int	29.5	unsigned short	>>=	_int	5.8
long	-=	long	28.9	unsigned char	<<=	_int	5.7
unsigned int	<<=	_int	28.3	other-types	%=	other-types	5.7
unsigned int	>>=	_int	28.2	long	+=	_int	5.6
unsigned int	^=	unsigned int	26.7	long	*=	_int	5.3
int	>>=	_int	26.2	unsigned long	&=	int	5.1
int	<<=	_int	25.5	unsigned long	/=	_int	5.0
int	/=	_int	23.8	unsigned int	&=	unsigned int	4.6
int	+=	int	22.1	unsigned int	|=	unsigned int	4.6
unsigned char	&=	int	19.7	long	%=	_int	4.6
unsigned int	&=	int	19.4	unsigned short	/=	_int	4.5
int	-=	int	17.4	unsigned char	&=	_int	4.3
long	^=	long	16.9	unsigned long	|=	_int	4.1
other-types	*=	other-types	16.8	unsigned char	|=	int	3.9
other-types	&=	other-types	16.7	long	<<=	_int	3.8
int	&=	int	16.2	float	*=	_double	3.7
unsigned long	<<=	_int	15.9	unsigned int	+=	unsigned int	3.5
other-types	^=	other-types	15.3	long	&=	int	3.5
other-types	/=	other-types	14.4	unsigned int	=	unsigned int	3.4
other-types	|=	other-types	13.5	int	%=	unsigned int	3.4
unsigned int	/=	_int	12.9	unsigned long	^=	int	3.3
ptr-to	+=	int	12.8	float	*=	double	3.3
unsigned int	%=	_int	12.6	unsigned long	*=	_int	3.1
int	%=	int	12.6	unsigned char	^=	unsigned char	3.1
int	=	int	12.3	unsigned char	^=	int	3.1
unsigned int	|=	_int	12.1	ptr-to	+=	unsigned long	3.1
float	*=	float	12.1	double	*=	double	3.1
int	|=	_int	12.0	unsigned short	/=	unsigned short	3.0
unsigned char	|=	_int	11.7	unsigned short	|=	int	3.0
unsigned int	%=	unsigned int	11.5	int	/=	unsigned int	3.0
unsigned char	%=	_int	11.5	float	/=	int	3.0
int	/=	int	11.4	double	/=	double	3.0
unsigned long	^=	unsigned long	11.3	unsigned int	+=	_int	2.9
int	^=	_int	11.1	float	*=	_int	2.9
int	=	_int	11.0	unsigned long	+=	unsigned long	2.8
unsigned char	>>=	_int	10.3	unsigned long	|=	unsigned long	2.8
other-types	>>=	other-types	9.6	unsigned long	|=	long	2.8
unsigned long	>>=	_int	9.5	long	=	long	2.8
int	*=	int	9.3	int	&=	_int	2.8
unsigned short	<<=	_int	8.9	float	=	float	2.8
unsigned int	*=	_int	8.4	unsigned int	-=	int	2.7
int	-=	_int	8.0	int	>>=	int	2.7
unsigned short	&=	int	7.9	int	^=	int	2.7
long	>>=	_int	7.7	unsigned char	=	_int	2.6
unsigned int	|=	int	7.5	float	-=	float	2.6
long	/=	_int	7.4	unsigned long	=	unsigned long	2.5
int	+=	_int	7.4	unsigned long	<<=	unsigned int	2.5
int	|=	int	7.4	int	<<=	int	2.5
unsigned short	%=	_int	6.9	float	/=	_double	2.5
other-types	<<=	other-types	6.7	int	*=	float	2.4
unsigned char	^=	_int	6.4	unsigned char	|=	unsigned char	2.3

For the other operators, each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator. 1311

Commentary

The discussion in the respective C sentences for these corresponding binary operators also apply here.

C++

5.15p7 *In all other cases, E1 shall have arithmetic type.*

Those cases where objects may not have some arithmetic type when appearing as operands to operators (i.e., floating types with the shift operators) are dealt with using the equivalence argument specified earlier in 5.15p7.

Semantics

compound assignment semantics

A *compound assignment* of the form **E1** *op* = **E2** differs from the simple assignment expression **E1** = **E1** *op* **(E2)** only in that the lvalue **E1** is evaluated only once. 1312

Commentary

This defines the term *compound assignment*. The original rationale for this form of operator was that it removed the need for translators to spot the commonly occurring case of the value of an object being operated on and stored back into the original object (an optimization that is made by translators for other languages).

expression 944 order of evaluation

There is no requirement that the evaluation order of E1 and E2 in the expression E1 op= E2 be the same as in the expression E1 = E1 op E2. In compiler terminology, E1 is known as a *common subexpression*.

Other Languages

Java specifies a left to right evaluation order and so in the following example the final values assigned to a and b is 9 in both cases. The C behavior in both cases is undefined.

```
1  void f(void)
2  {
3  short a = 6;
4  int   b = 6;
5
6  a += (a=3);    /* Undefined behavior. */
7  b = b + (b=3); /* Undefined behavior. */
8  }
```

In Java compound assignment operators perform a (narrowing primitive) conversion:

Java 15.26.2 *A compound assignment expression of the form* ***E1*** *op* ***= E2*** *is equivalent to* ***E1 = (T)((E1)*** *op* ***(E2))****, where* ***T*** *is the type of* ***E1****, except that* ***E1*** *is evaluated only once.*

This implicit conversions means that the expression a += b is conforming, while a = a + b generates a translation time diagnostic and an explicit cast (to short) has to be added.

Common Implementations

Because of the availability of these operators many translators make no direct effort to detect the optimizations possible in expressions such as x=x*y (some cases may be detected as a consequence of common subexpression detection). For the simple cases there is unlikely to be any differences in the generated machine code. In the more complex cases (e.g., x is an array element or structure member access) the opportunity to reuse the calculated address of an object is self-evident when a compound assignment operator is used and has to be deduced when the equivalent longer form is used.

Coding Guidelines

Compound assignment requires less effort to comprehend than its equivalent two operator form. Readers only need to comprehend two operands, and there is no need to notice that one operand is the same as another. These operators directly represent the concept of performing an operation on the value held in an object and storing it back into that object. Measurements of existing source shows that developers regularly use compound assignment operators. A guideline recommending this usage would serve no worthwhile purpose.

6.5.17 Comma operator

comma operator syntax

1313

expression:

assignment-expression
expression , *assignment-expression*

Commentary

This operator is frequently seen in automatically generated source code. It is usually used to simplify the analysis that needs to be performed by the generator in deducing what it has to do. For instance, when generating an expression it is not necessary to look ahead to see if some other expressions need to be evaluated first. If such an expression is encountered, it can occur as the left operand of a comma expression (which can occur in any context an expression can occur in).

Other Languages

The comma operator is unique to C (and C++). However, some languages provide a mechanism for grouping multiple statements into an expression that returns a value.

Common Implementations

compound expression

gcc supports what it calls *compound expressions*. The following example— the compound expression is delimited by ({ and }) and its value is that of the last expression in the compound— could be written using comma and conditional operators (however, it is not possible to define local objects within these operators).

```
1  extern int foo(void);
2
3  void f(void)
4  {
5  int loc;
6
7  loc =({
8       int x = foo (),
9           y;
10      if (x > 0)
11         y = x;
12      else
13         y = -x;
14      y;
15      });
16 }
```

Compound expressions do not offer much benefit in the visible source, except for automatically generated code. However, the ability to define objects can be useful in macro definitions. The following definition is written so that its arguments are only evaluated once.

```
1  #define MAXINT(x,y) ({int _x = (x), _y = (y); _x > _y ? _x : _y; })
```

Coding Guidelines

The comma token is one of the visibly smallest characters (in terms of display area occupied) and usually appears among other characters (unlike the semicolon token, which usually appears at the end of a line). Although specialist fonts have been designed for a variety of domains (e.g., mathematics, linguistics), none have yet become generally available for displaying source code. Increasing the visible size of the comma character is only one of several display issues that could be addressed.

Most occurrences of the comma token in the visible source are as a punctuator (e.g., an argument separator in function calls, or a list of identifiers in a declaration), not as an operator. The C syntax requires that any use of this token as an operator in an argument expression be enclosed in parentheses.

EXAMPLE 1319
comma operator

Semantics

---

comma operator left operand

The left operand of a comma operator is evaluated as a void expression; 1314

Commentary

The intent is for the left operand to generate side effects, but for any value it might have, not to take part in forming the result of the expression that contains it.

Coding Guidelines

cognitive 0 switch

The purpose of the left operand is to generate side effects without contributing to the value of the containing expression. As such it requires readers to make a cognitive task switch between comprehending the expression and updating their mental model (based on the consequences of the side effects). Is the cost/benefit to using a comma operator more worthwhile than separating its components over two statements (the semicolon after the first statement provides a sequence point)? There are three contexts in which an expression can appear:

1. *An initializer.* Initialization is a construct where C99 offers a solution not available in C90.

```
1  extern int glob;
2
3  void f(void)
4  {
5  int loc_1 = (--glob , glob + 3);
6
7  /*
8   * Only possible in C99.
9   */
10 glob--;
11 int loc_2 = glob + 3;
12
13 /*
14  * A possible rewriting in C90.
15  */
16 int dummy = glob--;
17 int loc_3 = glob + 3;
18 }
```

   Recommending that a comma operator in an initializer be replaced by a statement and a declaration limits portability by requiring a C99 translator. Also given that developers are not yet accustomed to seeing statements before declarations, it is possible that such statements will be overlooked.

2. *An expression statement.* There are no obvious benefits to using a comma operator in an expression statement. The single sequence point guarantee can be obtained using other constructs (although developers sometimes make incorrect assumptions about the sequencing of other operand evaluations).

comma operator 1315
sequence point

3. *A controlling expression.* The use of the comma operator in this context is discussed under the cases in which they can occur, selection statements and iteration statements.

selection statement 1739
syntax
iteration statement 1763
syntax

Until C99 translators become more generally available, the following guideline recommendation limits itself to expression statements.

Cg 1314.1 The comma operator shall not appear in the visible form of an expression statement.

If the left operand of the comma operator does not generate a side effect, it is redundant code. 190 redundant code

---

1315 there is a sequence point after its evaluation. comma operator sequence point

Commentary

This sequence point guarantees that any side effects that occur in the evaluation of the left operand will have taken place before the right operand is evaluated. There is no other guarantee given for the order of evaluation of any other operands that may occur within the full expression containing the comma operator.

C++

All side effects (1.9) of the left expression, except for the destruction of temporaries (12.2), are performed before the evaluation of the right expression. 5.18p1

The discussion on the function-call operator is applicable here. 1025 function call sequence point

Common Implementations

Sequence points cut down on the opportunities for optimization, or at least make them harder to find. Long stretches of source code without sequence points provide an optimizer maximum flexibility in ordering the sequence of generated machine code. For this reason translators prefer to evaluate the left operand of comma operators as soon as possible (in some case they may evaluate it as late as possible, but that is less commonly optimal). In:

```
1  extern int glob_1,
2             glob_2;
3  extern int g(void);
4
5  void f(void)
6  {
7  int loc[4];
8  /* ... */
9  loc[++glob_1] = (glob_2 * loc[0]) + (g() , glob_2) + (loc[3] * 3);
10 }
```

the only requirement placed on the call to g is that it occurs before the right operand of the comma operator is evaluated. Calling g after any of the other operands had been evaluated would require saving their intermediate results. Looking at the visible source code, there are obvious optimization advantages to calling g before any other operands are evaluated. But then a more detailed analysis may show that the other operands are already available in registers, and ready to be used, in which case it may be more efficient to call g as late as possible.

Coding Guidelines

An incorrect assumption sometimes made by developers is to assume that all operands to the left of a comma operator will be evaluated (and side effects have occurred) before any of the operands to its right are evaluated. This issue is covered by the guideline recommendation dealing with sequence points. 187.1 sequence points all orderings give same value

---

1316 Then the right operand is evaluated;

Commentary

The word *then* is sometimes incorrectly interpreted to mean that the right operand is evaluated immediately after the evaluation of the left operand; that is, no other operand is evaluated between the left and right operand evaluations. As the following example illustrates, other operands may be evaluated between the evaluations of the two operands of a comma operator.

```
1  extern int glob_1,
2             glob_2;
3  extern int g(void),
4             h(void); /* Writing extern int h(void); would be idiomatic. */
5
6  void f(void)
7  {
8  int loc_1;
9  /*
10  * After glob_1 is incremented there is no guarantee that g will be
11  * called before glob_2 is incremented, or that after glob_2 is
12  * incremented that h will be called before glob_1 is incremented.
13  */
14 loc_1 = (glob_1++ , g()) + (glob_2++ , h());
15 }
```

---

the result has its type and value.[95)] 1317

Commentary

The right operand has the same type and value as if it had not occurred as an operand of the comma operator.

If an attempt is made to modify the result of a comma operator or to access it after the next sequence point, the behavior is undefined. 1318

Commentary

This specification is consistent with the right operand being the result of a binary operator. However, it would also have been possible to treat this operand as if it had not been operated on, but the Committee chose not to make that decision.

C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue. lvalue 721

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++. lvalue 721

Common Implementations

Implementations are unlikely to use a temporary to store the value of the right operand. They will refer directly to the right operand.

Coding Guidelines

The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here.

Example

```
1  struct {
2          int m[10];
3         } x;
```

```
4
5   void f(void)
6   {
7   int *p = &((x , x).m[2]);
8
9   *p = 1;
10  p = &(x.m[2]);
11  *p = 1;
12  }
```

1319 EXAMPLE

EXAMPLE
comma operator

As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Commentary

Using an assignment operator in contexts other than an expression statement is discussed elsewhere.

1740 controlling expression
if statement

1320 95) A comma operator does not yield an lvalue.

footnote 95
comma operator
lvalue

Commentary

This is the one significant difference between an occurrence of the right operand on its own and as an operand of a comma operator. The former case could be an lvalue while the latter is not.

725 lvalue converted to value

C++

. . . ; the result is an lvalue if its right operand is. 5.18p1

```
1   #include <stdio.h>
2
3   void DR_188(void)
4   {
5   char arr[100];
6
7   if (sizeof(0, arr) == sizeof(char *))
8      printf("A C translator has been used\n");
9   else
10     if (sizeof(0, arr) == sizeof(arr))
11        printf("A C++ translator has been used\n");
12     else
13        printf("Who knows why we got here\n");
14  }
15
16  void f(void)
17  {
18  int loc;
19
20  (2, loc)=3; /* constraint violation */
21              // conforming
22  }
```

Forward references: initialization (6.7.8). 1321

6.6 Constant expressions

constant expression syntax

1322

```
constant-expression:
                conditional-expression
```

Commentary

constant expression not contain 1324

Syntactically specifying a *constant-expression* as a *conditional-expression* is consistent with constraints on operators that may appear in such an expression.

Coding Guidelines

How do people perform simple arithmetic? For operations involving small single-digit values, current models of human performance[60,817] are based on a network of connected arithmetic facts (e.g., $1 + 1 = 2$) stored in long-term memory. These *facts* are accessed by a process of spreading activation,[376,385] using as input the numbers and operators provided by the calculation. This process does not appear to require a person to have any understanding of the operation performed;[343] it is purely a fact-retrieval operation. People tend to use arithmetic procedures for larger values. The extent to which either a retrieval or procedural approach is used has been found to vary between cultures.[196] For a discussion of how people store arithmetic *facts* in memory, see Whalen.[1454] For a readable introduction to human processing of simple arithmetic quantities, see Dehaene.[338]

There have been some studies investigating people's eye movements while they performed simple arithmetic tasks.[1315] However, the operands in source code expressions are written on a horizontal line, not vertically under each other (as we are taught to do it in school). It is not known how this difference in operand layout will affect the sequence of eye movements that occur when performing simple arithmetic with source code expressions.

When a sequence of simple arithmetic operations are performed, the order in which they occur can affect the response time and error rate.[53] Other performance related behaviors include:

- *Problem size/difficulty.* Both the time taken to produce an answer and the error rate increase as the numeric value of the operands increases. It is thought that this effect occurs because operations on larger values are not as common as on smaller values; it is an amount-of-practice effect. Constants in C source code also tend to be small (see Figure 825.1).
- *Split effect.* People take longer to reject false answers that are close to the correct answer (e.g., $4\times7 = 29$) than those that are further away ($4\times7 = 33$).
- *Associative confusion effect.* Here people answer a different question from the one asked[543] (e.g., giving 12 as the answer to $4\times8 =?$, which would be true had the operation been addition).
- *Odd/even effect.*[834] Here people use a rule rather than retrieving a fact from memory to verify the answer to a question; for instance, adding an odd number to an even number produces an odd result.

developer errors memory overflow 0

The affect of working memory capacity limits on mental arithmetic performance is discussed elsewhere.

Description

A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be. 1323

Commentary

The operative phrase here is *can be*. The standard specifies a number of contexts where a numeric value is required at translation time. In all other cases a translator need only act as if the expressions are evaluated during translation. The standard also gives implementations freedom to evaluate other expressions at translation time. There is no abstract machine state available during translation, which limits what can be considered to be a constant expression.

1334 footnote 96
1344 constant expression other forms

Here the term *constant expression* refers to the syntactic form `constant-expression`, while *constant* refers to the syntactic form `constant`. Evaluation of constant expressions, by a translator, is often called *constant folding* (particularly in compiler writing circles).

822 constant syntax

C++

The C++ Standard says nothing about when constant expressions can be evaluated. It suggests (5,19p1) places where such constant expressions can be used. It has proved possible to write C++ source that require translators to calculate relatively complicated functions. The following example, from Veldhuizen,[1413] implements the pow library function at translation time.

```
template<int I, int Y>
struct ctime_pow
   {
   static const int result = X * ctime_pow<X, Y-1>::result;
   };

// Base case to terminate recursion
template<int I>
struct ctime_pow<X, 0>
   {
   static const int results =- 1;
   };

const int x = ctime_pow<5, 3>::result; // assign five cubed to x
```

Other Languages

Some languages do not support the use of operators in constant expressions; which are essentially literals. For instance, the first standard for Pascal, ISO 7185, did not; many implementations added such support as an extension and the first revision of that language standard added support for such usage. Other languages (e.g., Ada) allow the value of a constant to be calculated during program execution.

Common Implementations

There are two ways of representing integer constants during translation. A translator can either use the representation of the host on which the translator is executing, or it can use its own internal format (potentially often supporting a greater range of values). The latter approach has the advantage of offering consistent behavior, a benefit for those vendors offering products on a variety of hosts.

There is no context where the standard requires a translator to support arithmetic operations on floating constants during translation. Many implementations choose to generate code to evaluate expressions containing floating-point operands during program startup

Some translators and static analysis tools issue diagnostics for operations they consider to be suspicious; for instance, a left-shift operation that results in all of the bits set in the left operand being shifted out of the result (which can occur for a shift amount less than the width of the promoted left operand).

Flow analysis is used by some translators to deduce that particular uses of objects have a single value (known as *constant propagation*). For instance, if x appears in an expression immediately after the value 3 is assigned to it, a read of x can be replaced by 3. Deducing whether an expression has a unique constant value during program execution is undecidable[558,980] (even if the interpretation of conditional branches is ignored). Rather than deducing a unique constant value, *generalized constant propagation* attempts to

estimate the range of values an object may have at a particular point in the source. Verbrugge, Co, and Hendren[1416] compare the performance of various algorithms on C programs of under 1,000 statements.

Example

```
extern int glob;

void f(void)
{
glob = 4;

struct {
        /*
         * It may be possible to deduce the value of glob at translation
         * time, but the standard does not require such deduction.
         */
        int mem :glob; /* Constraint violation. */
       } loc;
glob++; /* Can be implemented by assigning 5 to glob. */
}
```

Constraints

constant expression not contain

Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[96)] 1324

Commentary

constant expression 1322 syntax

While the syntax does not permit a constant expression to contain an assignment or comma operator at the outermost level, it does not prohibit them from occurring within a parenthesized expression. This constraint specifies these operators cannot occur in any contexts that are evaluated within a constant expression. Because they are evaluated at translation time, an attempt to generate a side effect in a constant expression, could not be interpreted to have any meaning. The C abstract machine does not exist during translation.

abstract machine 184 C

C90

. . . they are contained within the operand of a **`sizeof`** *operator.*[53)]

sizeof 1119 result of

With the introduction of VLAs in C99 the result of the **`sizeof`** operator is no longer always a constant expression. The generalization of the wording to include any subexpression that is not evaluated means that nonconstant subexpressions can appear as operands to other operators (the logical-AND, logical-OR, and conditional operators). For instance, `0 || f()` can be treated as a constant expression. In C90 this expression, occurring in a context requiring a constant, would have been a constraint violation.

C++

Like C90, the C++ Standard only permits these operators to occur as the operand of a **`sizeof`** operator.

See C90 difference.

Other Languages

Languages that support constant expressions that are evaluated during program execution may also support the occurrence of side effects.

Coding Guidelines

As footnote 96 points out, it is possible for a subexpression not to be evaluated and still affect to the final value of the expression that contains it. A subexpression that does not affect the final value of the expression is not redundant code in the sense that it does not form part of the program image. An expression containing identifiers that are expanded by the preprocessor may have a generalized form that has been designed to handle a variety of different translation and host environments. 190 redundant code

In the following example, a particular constant may only be available when translating for a particular host. The generalized case involves a function call during program execution. A macro definition hides some implementation details behind the name Y; however, if it needs to be used in a context where a constant is required, more complexity may be needed. In this case another macro, X, and a conditional expression has been used.

```
#if defined VENDOR_A
   #define X 0
   #define Y VENDOR_A_VALUE
#else
   #define X NON_ZERO_DEFAULT_VALUE
   #define Y get_value_during_execution()
#endif

int glob[X ? X : Y];
```

Although a subexpression that does not affect the final value of an expression and does not contain identifiers that are macro expanded involves a cost and probably has no benefit; such uses are rare. For this reason no guideline recommending against using such constructs is given.

Example

```
extern int glob;
extern int g(void);

int f(void)
{
switch (glob)
   {
   case 0 && g() : return 77;

   case 1 || g() : return 88;

   case 1 ? 2 : g() : return 99;

   case sizeof(g()) : return 111;
   }
}
```

1325 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Commentary

If the evaluation of an expression, at execution time, would deliver a result that is not representable in its type, the behavior is undefined. The check on constant expressions can afford to be more rigorous because there is no execution-time penalty in translation-time checks on the evaluation of a constant expression. 947 exception condition

C++

The C++ Standard does not explicitly specify an equivalent requirement.

Example

```
1  #include <limits.h>
2
3  int a[INT_MAX * INT_MAX];
```

Semantics

An expression that evaluates to a constant is required in several contexts. 1326

Commentary

The standard lists many of the contexts requiring integer constant expressions (additional ones include the *constant-expression* in a designator and the expression that controls conditional inclusion). All initializers for objects having static storage duration must be constant.

footnote 1334 96
designator 1646
constant-expression
conditional 1869
inclusion
constant expression
initializer 1644
static storage duration object

Other Languages

A few languages do not require constants in any contexts. However, the execution-time overhead on such a language design decision is often perceived to be much higher than users are willing to tolerate. It also complicates the job of writing a translator.

Common Implementations

Extensions created by vendors sometimes relax the contexts in which a constant expression is required. A common extension is to allow nonconstant expressions in initializers for objects having file scope (also supported in C++).

Coding Guidelines

As discussed elsewhere, expressions that evaluate to constants in other contexts are sometimes suspicious.

controlling 1740
expression
if statement

If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment. 1327

Commentary

This is a requirement on the implementation. The prohibition against raising exceptions applies to floating constants, not to constant expressions.

floating 858
constant
conversion
not raise exception

To ensure adequate range and precision, a translator that evaluates floating expressions will need to be aware of the value of the FLT_EVAL_METHOD macro. Arithmetic range and precision are two attributes of floating-point expression evaluation. Others, not included by this requirement, are specified by the FLT_ROUNDS macro and the FLT_RADIX macro. Also, while the status of FENV_ACCESS macro is known to the implementation, the affect of any execution-time calls to the floating-point status library functions will not.

FLT_EVAL_METHOD 354
FLT_ROUNDS 352
FLT_RADIX 366

C++

This requirement is not explicitly specified in the C++ Standard.

Other Languages

Very few languages require floating expressions to be evaluated during translation.

Common Implementations

A few implementations (e.g., gcc) do evaluate floating expressions during translation. Translators may, or may not, provide an option to control the optimization of floating-point operations (gcc provides many).

---

integer constant expression

An *integer constant expression*[97] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, and floating constants that are the immediate operands of casts. 1328

Commentary

This defines the term *integer constant expression*.

The restriction on floating constants only being the *immediate operands of casts* means that translators are not required to perform arithmetic operations on floating constants. Evaluating the result of casting of a floating-point constant to an integer type can be achieved by manipulating sequences of characters, starting with the initial token (no numeric representation of the floating constant need be created).

For the result of the **sizeof** operator to be an integer constant, its' operand cannot have a VLA type.

C++

> . . . , ***const*** *variables or static data members of integral or enumeration types initialized with constant expressions (8.5),* . . . 5.19p1

For conforming C programs this additional case does not cause a change of behavior. But if a C++ translator is being used to develop programs that are intended to be conforming C, there is the possibility that this construct will be used.

```
1  const int ten = 10;
2
3  char arr[ten]; /* constraint violation */
4                 // does not change the conformance status of the program
```

Other Languages

Some languages regard enumeration and character constants as different types and would require a cast for such constants to appear as operands in an integer constant.

Common Implementations

Some implementations map floating constant tokens to numeric values before casting them to an integer type.

Coding Guidelines

Any reader comprehension issues apply whether or not an expression can be evaluated to a constant; for instance, use of parentheses. 943.1 expression shall be parenthesized

Example

```
1  /*
2   * Both declarations define an array containing one element.
3   */
4  static int arr_1[2u + -1];
5  static int arr_2[1 + (int)0E99999999999999999999999999999999999999999];
```

1329 Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **sizeof** operator. integer constant cast of

Commentary

Supporting casts to a floating type would require translators to be capable of handling floating-point arithmetic (unless the only subsequent operation on the result was a cast to an integer type). Although address constants are supported, the model used is one where they are handled at link-time. This means their value is not available, during the translation of a source file, to be cast to an integer constant. 1341 address constant

The evaluation of the **sizeof** operator involves deducing the type of its operand, which does not necessarily involve evaluating any of the contained subexpressions.

Common Implementations

A few implementations (e.g., `gcc`) support casting constants to floating types.

More latitude is permitted for constant expressions in initializers. 1330

Commentary

program startup 150

The C language specification of initializers does not require translation-time knowledge of their values. The initialization can be evaluated as part of program's startup, allowing translators to treat initializers like any other expression. For instance, a translator need not concern itself with how many digits are significant in the evaluation of:

```
1  int x = (int)(1.0/3.0 + 2.0/3.0);
```

C++

5.19p2 *Other expressions are considered constant-expressions only for the purpose of non-local static object initialization (3.6.2).*

Other Languages

Most languages that support the use of initializers in declarations allow noninteger types to be assigned values having the appropriate type.

Such a constant expression shall be, or evaluate to, one of the following: 1331

Commentary

initializer type constraints 1660

As well as evaluating to one of the constructs on this list, the type of the constant expression also has to be compatible with the type of the object it is initializing.

— an arithmetic constant expression, 1332

Commentary

arithmetic constant expression 1339

This case is discussed elsewhere.

---

constant null pointer constant

— a null pointer constant, 1333

Commentary

null pointer constant 748

The standard specifies two possible token sequences that can be used to represent the null pointer constant.

---

footnote 96

96) The operand of a **`sizeof`** operator is usually not evaluated (6.5.3.4). 1334

Commentary

sizeof result of 1119
sizeof operand evaluated 1122

The only time the operand of a **`sizeof`** operator may need to be evaluated is if its operand contains a VLA. If the operand of the **`sizeof`** operator contains a VLA, it is not a constant expression and must be evaluated at execution time.

C90

*The operand of a **`sizeof`** operator is not evaluated (6.3.3.4) and thus any operator in 6.3 may be used.*

Unless the operand contains a VLA, which is new in C99, it will still not be evaluated.

C++

The operand is never evaluated in C++. This difference was needed in C99 because of the introduction of variable length array types.

1335 97) An integer constant expression is used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant.

footnote 97

Commentary

This is a partial list of all places where the standard requires an integer constant expression (arrays having block scope no longer require their size to be specified using an integer constant expression).

Other Languages

Some other languages require translation-time constant expressions (in some cases constants) in these contexts, while others support execution-time values in these contexts (or their equivalent forms).

1336 Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

Commentary

These constraints are a result of the preprocessor having less information available to it and are discussed elsewhere.

1869 conditional inclusion constant expression

1337 — an address constant, or

Commentary

This is discussed elsewhere.

1341 address constant

1338 — an address constant for an object type plus or minus an integer constant expression.

Commentary

An algorithm may call for the initialization value of the object being defined, with pointer type, to point to an element of an object other than its first. While recognizing the developer's desire for such functionality the Committee was aware that not all linkers were capable of providing it. They also recognized the commercial impracticality of requiring vendors to provide a linker to replace the one provided by the vendor of the host environment. It was believed that the majority of existing linkers supported a method of adding constant offsets to an existing address. This C sentence reflects this availability.

The requirement that the result of pointer arithmetic still point within (or one past the end) of the original object still apply.

1169 pointer one past end of object

C++

The C++ language requires that vendors provide a linker for a variety of reasons; for instance, support for name mangling.

Other Languages

Some languages effectively support such initializers because they allow the address of an element of an array or member of a structure to be used as an address constant.

Common Implementations

An implementation's support for additional forms of address constant is likely to be dictated by the functionality available in the linker used.

Example

```
1  extern int glob;
2
3  static int *p_g = &glob + (sizeof(int) * 2);
```

---

arithmetic constant expression

An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **sizeof** expressions. 1339

Commentary

integer constant expression 1328

This defines the term *arithmetic constant expression*. An arithmetic constant expression is a more general kind of constant expression than an integer constant expression in that the restrictions on operands having a floating type have been lifted. Annex F, of the C Standard, also discusses arithmetic constant expressions.

Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **sizeof** operator whose result is an integer constant. 1340

Commentary

integer constant cast of 1329

This specification is a more generalized form of the one given for integer constant expressions. It includes support for the conversion of arithmetic types to floating types.

---

address constant

An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; 1341

Commentary

This defines the term *address constant*. The term *scalar constant expression* would not be appropriate because arithmetic types are not included in the list. Common implementation practice (also for other languages) is for information on the storage address of objects having static storage duration and the address of the start of a function definition to be included in a program's image. This C definition of address constant reflects this existing implementation practice.

While objects having static storage duration are represented in a fixed, unique storage location, objects having other storage durations can be allocated storage at different locations every time their lifetime starts; the address of objects having automatic storage duration need not be unique (because of recursive invocations of the function that contains them).

The only context in which an address constant is required is the initializer for an object having a pointer type.

C90

The C90 Standard did not explicitly state that the null pointer was an address constant, although all known implementations treated it as such.

C++

The C++ Standard does not include (5.19p4) a null pointer in the list of possible address constant expressions. Although a null pointer value is listed as being a *constant-expression* (5.19p2). This difference in terminology does not appear to result in any differences.

Other Languages

Many languages only permit pointers to point at dynamically created objects. Such languages cannot contain address constants.

Common Implementations

program startup 150

In a hosted environment the relative addresses of objects, with static storage duration, and developer-written function definitions is usually decided by the linker. On program startup an area of storage is allocated to hold the objects having static storage duration. A fixed offset is added to the relative addresses of objects to obtain their actual addresses in the program's address space.

Library functions may be dynamically linked and their actual (possibly virtual) address is not known until they are called during program execution. In this case translators need to generate the machine code needed to ensure this execution-time address fix-up occurs.

In a freestanding environment the actual storage locations (addresses) assigned to objects (with static storage duration) and the machine code representing function definitions are usually known at link-time. For the other cases the complexity of program loading varies from always using a known fixed offset to the full virtual memory handling seen in hosted environments.

1342 it shall be created explicitly using the unary **&** operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type.

Commentary

These uses of the unary **&** operator are also techniques for generating addresses in nonconstant contexts. This list does not include using the value of an object to create an address (even if that object was assigned an address constant earlier in the same translation unit), or the value returned from a function call. Casts of integer constants to pointer types are often required in programs executing in freestanding environments, where it is known that certain addresses are mapped to hardware devices.

752 integer permission to convert to pointer

C90

Support for creating an address constant by casting an integer constant to a pointer type is new in C99. However, many C90 implementations supported this usage. It was specified as a future change by the response to DR #145.

C++

Like C90, the C++ Standard does not specify support for an address constant being created by casting an integer constant to a pointer type.

Other Languages

Expressions having array or function type are not usually implicitly converted to addresses in other languages. Support for casting integer constants to addresses varies between languages.

Common Implementations

Many implementations support, as an extension, the casting of address constants to different pointer types as also being address constants.

Example

```
1  extern int glob;
2  int *g_p = (int *)1;
3  char *g_c = (char *)&glob;
```

1343 The array-subscript **[]** and member-access **.** and **->** operators, the address **&** and indirection ***** unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

Commentary

The array-subscript **[]** and member-access **.** and **->** operators can all be used to specify particular subobjects within objects.

The operand used as the index in the array-subscript **[]** operator must be an integer constant expression if the resulting expression is to be an address constant.

The indirection ***** unary operator would normally be used to access the value of an object. However, when used in combination with the address **&** operator, the access can be *canceled out*.

1092 &*

The member-access -> operator requires a pointer type for its left operand. This will need to be an address constant if the resulting expression is to be an address constant.

Casts to structure type are not permitted in constant expressions, but casts of integer constants to pointer-to structures are permitted. A common implementation of the `offsetof` macro is:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

Example

```
struct S {
          int m_1;
          short m_2[3];
         } x, y;

struct S *p1_s = &x;
int *p1_m1_s = &(y.m1);
int *p2_m1_s = &((&y)->m1);
int *p3_m1_s = &(((struct S *)0)->m1);
int *p4_m1_s = &(&*y.m1);

int arr[10];
int *p1_arr = arr;
int *p2_arr = &(arr[4]);

short *p3_m2_s = y.m_2;
short *p4_m2_s = &(y.m_2[0]);
```

---

constant expression other forms

An implementation may accept other forms of constant expressions. 1344

Commentary

Here the standard gives explicit permission for translators to extend the kinds of expressions that are translation-time constants.

C++

The C++ Standard does not given explicit latitude for an implementation to accept other forms of constant expression.

Other Languages

Translators for other languages sometimes include extensions to what the language specification defines as a constant expression. Not many language definitions explicitly condone such extensions.

Coding Guidelines

Developers may be making use of implementation-defined constant expressions when they use one of the standard macros (e.g., invoking the `offsetof` macro). In this case the choice is made by the implementation and is treated as being invisible to the developer (who may be able to make use of a translator option to save the output from the preprocessor to view the constant). The guideline recommendation dealing with the use of extensions is applicable here.

extensions 95.1 cost/benefit

Example

```
static int foo = 2;
static int bar = foo + 2; /* A constant in the Foobar, Inc. implementation. */
```

1345 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.[98)]

constant expression semantic rules
1869 conditional inclusion constant expression

Commentary

Additional constraints on constant expressions apply when they appear in conditional inclusion directives.

C++

The C++ Standard does not explicitly state this requirement, in its semantics rules, for the evaluation of a constant expression.

Common Implementations

Because the evaluation occurs during translation rather than during program execution, the effects of any undefined behaviors may be different. For instance, a translator may perform checks that enable a diagnostic to be generated, warning the developer of the conformance status of the construct.

1346 **Forward references:** array declarators (6.7.5.2), initialization (6.7.8).

1347 98) Thus, in the following initialization,

footnote 98

```
static int i = 2  || 1 / 0;
```

the expression is a valid integer constant expression with value one.

Commentary

While this usage is unlikely to appear in the visible source, as such, it can occur indirectly as the result of macro replacement.

C++

The C++ Standard does not make this observation.

6.7 Declarations

declaration syntax

1348

declaration:
declaration-specifiers *init-declarator-list*$_{opt}$;
declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$
function-specifier *declaration-specifiers*$_{opt}$
init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*
init-declarator:
declarator
declarator = *initializer*

Commentary

The intent of this syntax is for an identifier's declarator to have the same visual appearance as an instance of that identifier in an expression. For instance, in:

```
1  int x[3], *y, z(void);
2  char (*f(int))[];
```

the identifier `x` might appear in the source as an indexed array, `y` as a dereferenced pointer, and `z` as a function call. An example of an expression using the result of a call to `f` is `(*f(42))[1]`.

C90

Support for *function-specifier* is new in C99.

C++

The C++ syntax breaks declarations down into a number of different categories. However, these are not of consequence to C, since they involve constructs that are not available in C.

The nonterminal *type-qualifier* is called *cv-qualifier* in the C++ syntax. It also reduces through *type-specifier*, so the C++ abstract syntax tree is different from C. However, sequences of tokens corresponding to C declarations are accepted by the C++ syntax.

The C++ syntax specifies that *declaration-specifiers* is optional, which means that a semicolon could syntactically be interpreted as an empty declaration (it is always an empty statement in C). Other wording requires that one or the other always be specified. A source file that contains such a construct is not ill-formed and a diagnostic may not be produced by a translator.

Other Languages

Some languages completely separate the syntax of the identifier being declared from its type (requiring the type to appear to the left of the identifier being declared, or on the right). For instance, in the Pascal declaration:

```
1  x : array[1..3] of integer;
2  y : ^ integer;
3  function z() : integer; (* Function types don't follow the pattern. *)
```

the identifiers `x`, `y`, and `z` are separated from their type by a colon.

In a few languages the identifier being declared is also part of a declarator, which may include type information (e.g., an example array declaration in Fortran is `Integer Arr(3)`). Java syntax supports two methods of declaring objects to have an array type.

```
1  int [] a1,
2         a2[];
3  int b1[],    // has the same type as a1
4      b2[][];  // has the same type as a2
```

Ada does not allow object declarations to occur between function definitions. The intent[619] was to avoid the poor readability that occurs when items that are visibly smaller, because they contain few characters (an object declaration) are mixed with items that are visibly larger, because they contain many characters (a function definition). However, this restriction is cumbersome and was removed in Ada 95.

Common Implementations

object specifying address

Some implementations (e.g., most vendors target freestanding environments) support an extension that enables developers to specify the start address to be used for the object or function defined by the declaration. The token `@` is often used.

```
1  int i @ 0x800;
```

`gcc` supports the **`__attribute__`** in declarations. Quite a large number of different attributes are supported (an identifier denoting the attribute appears between a pair of parentheses). They can apply to functions, objects, or types. For instance, `int square (int) __attribute__((const));` specifies an attribute of the function `square`.

Coding Guidelines

The visual layout of declarations can affect the reader effort required to extract information and the likelihood of them making mistakes. Declaration layout has three degrees of freedom:

declarations in which source file 1810

1. The source file selected to contain the declaration. This issue is discussed elsewhere.

2. The order in which different declarations occur within the same file or block (individual declarations can often occur in many different orders with the same effect). Identifiers declared at file scope are often read by people having more diverse information-gathering needs than those declared at block scope (where the use is usually very specific). Also, long sequences of identifier declarations usually only occur, with any frequency, at file scope. The number of declarations a block scope tend to be small. The quantity of file scope declarations and the diverse needs of readers suggests that an investment in organizing these declarations and providing more detailed commenting will produce a worthwhile benefit.

286 identifiers number in block scope

3. The layout of a single declaration and the token sequence used to specify it. (It is often possible to use different token sequences to specify the same declaration.)

The ordering of declarations at file scope

There are a number of patterns, in declaration ordering, commonly seen in existing source code:

- **`#include`** preprocessing directives are invariably placed before most other declarations in a file. This directive occupies a single line and placing all of them first ensures that subsequent declarations will be able to reference the identifiers they declare.
- All declarations of macros, typedefs, objects, and function declarations usually occur before the function definitions. The issue of ordering function definitions is discussed elsewhere.

1821 function definition syntax

- The grouping of declarations of macros, typedefs, objects, and function declarations. This grouping may be by kind of declaration (e.g., all macros, all typedefs, etc.), or it may be by some internal subdomain (e.g., all declarations related to a tree data structure, or the organization of program output, etc.).
- New declarations of a particular kind of identifier (e.g., macro or object) are often added at the end of the list of existing declarations of those kinds of identifiers.

Peoples performance in recalling, recently remembered, words from a list has been found to have a number of characteristics. Studies by Howard and Kahana[597, 598] investigated the probability of a particular word on a list being recalled when subjects were given the name of another word. The results showed (see Figure 1348.1) that the words immediately following the words presented were most likely to be recalled.

When subjects were asked to perform free recall of a list of words they had seen, successive recalled words tended to be related in some way (responses could be given in any word order) rather than being completely unrelated. A measure of pairwise similarity between the words presented was calculated using latent semantic analysis. The results showed that these LSA similarity values were highly predictive of the order in which subjects recalled words.

792 latent semantic analysis

What are the costs and benefits of putting declarations in a particular order? As discussed elsewhere developers tend to read source on an as-needed basis. A developer is unlikely to need to read a declaration unless it declares an identifier that is referenced from other source that is currently being read. Reading an identifier's declaration, once found, may in turn create the need to read other identifier declarations.

770 reading kinds of

To locate an identifier using a text editor, a developer might go to the top of the file and use a search command to find the first occurrence. Within an IDE, it is often possible to click on an identifier to bring up a window containing its declaration. A developer might also perform a quick visual search.

121 IDE

If the information required relates to a single identifier, the cost is the search time for that identifier declaration. The cost of obtaining information on N identifiers can be much greater than N times the search cost of a single identifier declaration. Reading a declaration can be like walking a tree— one declaration leading to another and then back to the original for more information extraction.

Having all related declarations visible on the screen at the same time (perhaps in multiple windows) minimizes the keyboard and mouse activity needed to switch between different declarations.

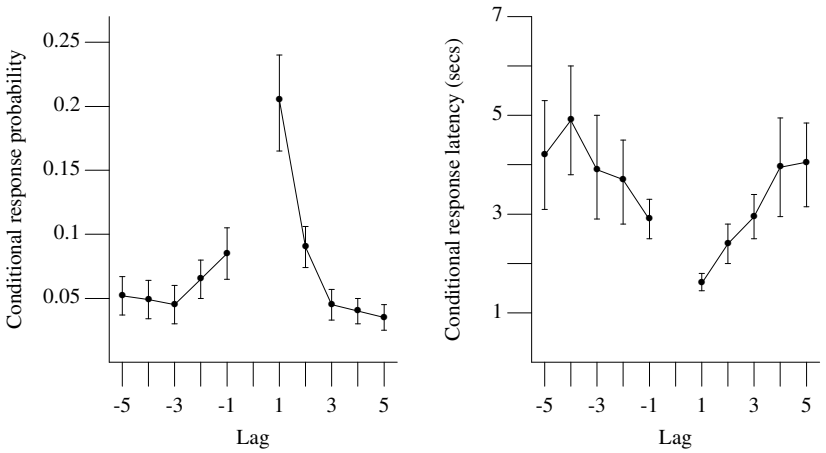


Figure 1348.1: The *lag recency effect*. The plot on the left shows the probability of a subject recalling an item having a given lag, while the plot on the right gives the time interval between recalling an item having a given lag (error bars give 95% confidence interval). If a subject, when asked to remember the list "ABSENCE HOLLOW PUPIL", recalled "HOLLOW" then "PUPIL", the recall of "PUPIL" would have a lag of one ("ABSENCE" followed by "PUPIL" would be a lag of 2). Had the subject recalled "HOLLOW" then "ABSENCE", the recall of "ABSENCE" would be a lag of minus one. Adapted from Howard and Kahana.[598]

How can declarations be ordered to minimize total cost? Answering this question requires usage information. What percentage of searches involve locating a single identifier declaration and what percentage involve reading multiple declarations? To what extent do related declarations require readers to visually skip back and forth between them? Unfortunately, this usage information is not available.

Ordering declarations alphabetically, based on the identifier declared, may improve the performance of some identifier search strategies (e.g., paging up/down until the appropriate sorted identifier is reached). However, such an ordering may be difficult to achieve in practice because of the need to declare an identifier before it is referenced and because of the complications introduced by those declarations that contain more than one identifier (e.g., enumeration types). Also there is nothing to suggest how large improvements in search performance will be, or whether it will give alphabetic ordering the edge over other orderings. Alphabetic ordering of identifier declarations is not discussed further here.

The following discussion assumes that for all ordering of declarations, developer search time is approximately the same. The important factor then becomes the ease with which developers can view multiple, related declarations.

If there is a dependency between two declarations, it is likely that a developer will want to read both of them. Placing dependent declarations sequentially next to each other in the visible source could reduce the need to switch between views of them (using the keyboard or mouse to change display contents). Both declarations being visible on the display at the same time. However, for all but the simplest declarations, placing dependent declarations sequentially next to each other is rarely possible:

- There may be many references to a particular identifier. How should the many declarations making these references be ordered?
- There may be many identifiers referred to in one declaration. For instance, the members of a structure type may have a variety of types. How would such multiple referenced identifiers be ordered?

categorization

Looking at existing source shows that in many cases declarations are grouped by categories (categorization is discussed in the Introduction). The chose of categories, by developers, seems to have been based on a variety of reasons, including:

- All objects declared to have the same type

- All function declarations
- Object declarations holding semantically related information (where it was not considered to worthwhile create a structure type to hold them)
- Objects having static linkage
- all declarations related to some data structure; for instance, types used in the manipulation of tree data structures may include one or more structure types, perhaps a union type, and a variety of enumeration or macro definitions (giving symbolic names to the numeric values used to denote various kinds of tree nodes)
- Grouped in the order in which they were originally written, which is likely to have been influenced by the order (both in time and space, within the file) in which function definitions were written

822 symbolic name

Placing declarations in the same category adjacent to each other is using an ordering relationship. (Alphabetic ordering is another one.) It might be claimed that category ordering is an effective heuristic for minimizing the developer cognitive and motor cost of switching between different views of the source while attempting to comprehend a declaration. However, there is no evidence to support this claim, or indeed any other claims of cost-effective declaration orderings.

Visual layout of adjacent and single declarations

declaration visual layout

It is quite common to see sequences of object declarations where the identifiers being declared are vertically aligned with each other. (A number of rationales can be heard for having such an alignment: ease of readability and "it looks nice" are both frequently given.) This discussion concentrates on the visual issues of adjacent identifier declarations. Vertical alignment creates an edge. When scanning the source, edges are something that readers can use to help control what they look at.

770 Edge detection
770 reading kinds of

The gestalt principle of continuation suggests that aligning identifiers creates an association between them. The principle of similarity (types in one set, objects being declared in the other) does not occur visually, although it may exist as a category in the reader's mind. Do we want to associate identifiers from different declarations with each other? The issue of these associations and reader categories are discussed elsewhere.

770 gestalt principles
770 continuation gestalt principle of
1358 declarator list of

Many declarations declare a single identifier on each source line. Developers often visually search lists of declarations by scanning up or down those declarations. Consequently, when a second identifier is declared on the same line, it is sometimes missed. As the following example shows, the surrounding declarations play an important role in determining how individual identifiers stand out, visually. A reader looking for the identifiers `c` or `total_count` is likely to notice that more than one declaration occurs on the same line. However, a search for `zap` may fail to notice it because of the degree to which its declaration blends in to give the appearance of a single declaration (perhaps `zip_zap`).

```
1  int a, b, c;
2
3  int foo_bar,
4      zip,zap,
5      win_lose;
6
7  int local_val,
8      max_count, total_count,
9      global_val;
10
11 void j(int k, int l);
```

A guideline recommendation based on visual appearance would be difficult to word and very difficult for developers to judge and for tools to enforce. The simple solution is to have a simple rule covering all cases (accepting that in some situations it is redundant).

Cg 1348.1 No more than one *init-declarator* shall occur on each visible source code line.

What about identifiers appearing in other kinds of declarations?

- *Preprocessor syntax* only permits one macro to be defined per line.
- *Function declarations.* The function name usually appears in the same location as it would in an object declaration (having the same type as the function return type). Additional information is provided by the parameter declarations. The parameter information is usually only read after the function name has been located; it is rarely searched for independently of the function name. Given the localized nature of visual searches of parameter information, there is no reason to split these declarations across multiple lines.
- *Function definitions.* The identifiers declared in a function definition's parameter list need to be included in any search of locally defined objects.

enumeration 517
set of named constants

- *Members of enumeration types.* This issue is discussed elsewhere.

struct-declarator 1390.1
one per source line

- *Members of structure and union types.* This issue is discussed elsewhere.

Cg 1348.2

For the purposes of visual layout the parameters in a function definition shall be treated as if they were *init-declarator*s.

type specifier 1378
syntax

The relative ordering of some tokens within a declaration is variable. The syntax does not impose any relative ordering on type specifiers, type qualifiers, or storage-class specifiers. This issue is discussed in elsewhere.

An object having static storage duration in block scope is rare. Whether there is a worthwhile cost/benefit in drawing attention to such a definition is open to debate.

identifier definition close to usage
grouping 1707
spatial location

There are benefits in visually grouping related items near each other. However, since it is often possible to comprehend statements without detailed knowledge of the identifiers they contain, the cost difference of them not being visually close may be small. Given that approximately 70% of block scope declarations (excluding parameters) occur in the outermost block scope (see Figure 408.1) and many function definitions are short enough (see Figure 1707.9) to be completed viewed on a display, the applicable definition may already be easy to locate (many identifiers are not declared within the minimum block scope required by their usage, Figure 190.1).

There is cost associated with declaring identifiers within the minimum required scope, compared to declaring them in the outermost block. When writing the body of a function definition a developer may be unsure of the minimal required scope of some identifiers. Developers might chose to move declarations on an as-needed basis or might choose to move declarations to the appropriate scope once the body is finalized. Whichever choice is made there is a cost that needs to be paid. Also during maintenance the required scope may change. This cost of deducing and maintaining local declarations in a *minimal* scope appears to be greater than the benefits.

Example

```
1  typedef int foo,
2              bar;
3
4  void f(void)
5  {
6  foo(bar);               /* Declare bar as an object.   */
7  extern int foo(void);   /* Declare foo as a  function. */
8  }
```

Table 1348.1: Occurrence of types used in declarations of objects (as a percentage of all types). Adapted from Engblom[389] and this book's benchmark programs.

Type	Embedded	Book's benchmarks
integer	55.97	37.5
float	0.05	1.6
pointer	22.08 (data)/0.23 (code)	48.2
struct/union	9.88	6.1
array	11.80	6.6

Table 1348.2: Occurrence of types used to declare objects in block scope (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`int`	28.1	`long`	3.0
`struct *`	27.7	`union *`	2.9
other-types	10.8	`unsigned short`	2.3
`unsigned int`	5.5	`unsigned char`	2.0
`struct`	4.9	`char`	1.8
`unsigned long`	4.8	`char []`	1.5
`char *`	3.5	`unsigned char *`	1.3

Table 1348.3: Occurrence of types used to declare objects with internal linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`int`	20.9	`const char []`	2.4
other-types	14.4	`unsigned int`	1.8
`struct`	13.0	`const struct`	1.8
`struct *`	8.2	`void *()`	1.7
`struct []`	7.4	`const unsigned char []`	1.6
`( const char * const ) []`	4.0	`unsigned int []`	1.4
`unsigned char []`	3.4	`int *()`	1.4
`unsigned short []`	3.3	`( struct * ) []`	1.3
`int []`	2.9	`( char * ) []`	1.3
`char *`	2.8	`unsigned long`	1.2
`char []`	2.7	`const short []`	1.2

Table 1348.4: Occurrence of types used to declare objects with external linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`int`	22.8	`char *`	3.2
`const char []`	15.4	`union *`	3.0
other-types	10.6	`enum`	2.4
`struct *`	10.3	`float`	1.4
`const struct`	10.2	`char []`	1.4
`struct`	8.2	`unsigned int`	1.2
`void *()`	4.6	`int []`	1.2
`struct []`	4.1		

Constraints

declaration shall declare identifier

A declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration. 1349

Commentary

It is C's unusual declaration syntax that makes it possible to write a declaration that declares no identifier in the scope of the declaration. A declaration that did not declare a declarator might be regarded as simply redundant. However, such usage is suspicious; it is likely that the developer has made a mistake. This wording requires that a declaration declares an identifier that is visible in the scope of the declaration.

redundant code 190

C90

6.5 *A declaration shall declare at least a declarator, a tag, or the members of an enumeration.*

The response to DR #155 pointed out that the behavior was undefined and that a diagnostic need not be issued for the examples below (which will cause a C99 implementation to issue a diagnostic).

```
1  struct { int mbr; }; /* Diagnostic might not appear in C90. */
2  union { int mbr; };  /* Diagnostic might not appear in C90. */
```

Such a usage is harmless in that it will not have affected the output of a program and can be removed by simple editing.

Other Languages

The declaration syntax of most languages does not support a declaration that does not declare an identifier.

Example

```
1  struct {
2          int m1; /* Constraint violation. */
3         };
4  struct {
5          struct T {int m11;} m1; /* Declares a tag. */
6         };
7
8  typedef enum E { E1 }; /* Pointless use of a typedef. */
9
10 long int; /* No declarator. */
11
12 ; /* Syntactically a statement. */
```

declaration only one if no linkage

If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3. 1350

Commentary

linkage 421 kinds of

type 1454 contents defined once

Identifiers with no linkage include those having block and function prototype scope, those in the label name space, and typedef names. The special case for tags is needed to handle recursive declarations, which are discussed elsewhere.

C++

C++ one definition rule

This requirement is called the *one definition rule* (3.2) in C++. There is no C++ requirement that a typedef name be unique within a given scope. Indeed 7.1.3p2 gives an explicit example where this is not the case (provided the redefinition refers to the type to which it already refers).

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

```
typedef int I;
typedef int I; // does not change the conformance status of the program
               /* constraint violation */
typedef I I;   // does not change the conformance status of the program
               /* constraint violation */
```

Other Languages

Some languages only allow one declaration of the same identifier in the same scope and name space. Those that require declaration before usage and support recursive data structures or recursive function calls need to provide a *forward* declaration mechanism (Pascal actually uses the keyword **forward**). Use of this mechanism effectively creates two declarations of the same identifier.

1351 All declarations in the same scope that refer to the same object or function shall specify compatible types.

declarations refer to same object
declarations refer to same function

Commentary

Occurrences of multiple declarations specifying incompatible types are probably the result of some developer mistake. Specifying that an implementation choose one of them is unlikely to be of benefit to developers, while generating a diagnostic pointing out the usage enables the developer to correct the mistake.

C++

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.

3.5p10

C++ requires identical types, while C only requires compatible types. A declaration of an object having an enumerated type and another declaration of the same identifier, using the compatible integer type, meets the C requirement but not the C++ one. However, a C++ translator is not required to issue a diagnostic if the declarations are not identical.

```
enum E {E1, E2};

extern enum E glob_E;
extern int glob_E; /* does not change the conformance status of the program */
                   // Undefined behavior, no diagnostic required

extern long glob_c;
extern long double glob_c; /* Constraint violation, issue a diagnostic message */
                           // Not required to       issue a diagnostic message
```

Other Languages

Languages that support some form of duplicate type declarations usually require that the types be the same.

Coding Guidelines

Adhering to the guideline recommendation specifying the placement of declarations of identifiers having file scope in a single file helps ensure this constraint is met.

422.1 identifier declared in one file

Semantics

1352 A declaration specifies the interpretation and attributes of a set of identifiers.

declaration interpretation of identifier

Commentary

value 73

For instance, the type of an object determines how its contents are to be interpreted. Attributes of the declaration of an object might be its storage class, while the textual location of the declaration decides the scope of the identifiers it declares.

Other Languages

Some languages do not require that identifiers appear in a declaration. Their interpretation and attributes are deduced from the context in which they occur and the operations performed on them. In other languages (e.g., Visual Basic, Perl, and Scheme) the type information associated with an object can change depending on the last value assigned to it. Values have type and this type information is assigned to an object, along with the value. Values are said to be tagged with their type. (The Scheme revised report[719] refers to the language as using *latent* types, while other terms include *weakly* or *dynamically* typed.) BCPL requires that identifiers be declared, but deduces their type from the usage.

Common Implementations

Extensions to C declarations often include creating additional specifiers that can appear in a declaration. For instance, the keywords **`near`**/**`far`**/**`huge`** used to specify how a pointer is to be interpreted.

Coding Guidelines

naming conventions 792

The issue of encoding information on some of these attributes is discussed elsewhere.

Judging whether an identifier is used in ways that involve different sets of semantic associations is something that is not yet possible, in general, using an automated tool.

object role

Some coding guideline documents recommend that instances of particular objects, within a program, only be used for a single purpose. For instance, as a loop counter, or to hold the running total of some quantity. Using the same object for both purposes, assuming the two uses do not overlap in the source code, may reduce the total amount of storage used but experience shows that it generally increases the effort needed to comprehend the code and the likelihood of faults being introduced.[1352.1]

boolean role 476
bit-set role 945

The *purpose* of an object is rather difficult to pin down. However, the way in which an object is used (its *role* in the source) often follows a pattern that can be categorized. The role categories discussed in other coding guideline subsections are based on the kinds of values an object might have (e.g., boolean and bit-set roles) and the operators that might be applied to them. Sajaniemi[1191] proposed a role classification scheme based on an analysis of a higher level of abstraction of how objects are used in the code. For instance:

- *constant*: an object whose value does not change after initialization,
- *stepper*: an object that iterates through a series of values that do not depend of the values of other nonconstant variables,
- *follower*: an object that iterates through a series of values that depended on the value of a stepper variable,
- *most-recent holder*: an object holding the latest value encountered during the processing of a sequence of values,
- *most-wanted holder*: an object holding the best value encountered during the processing of a sequence of values,
- *gatherer*: an object whose value represents the accumulated affect (e.g., the sum) of individual values encountered during the processing of a sequence of values,
- *one-way flag*: an object whose value is changed once, while processing some sequence of values,
- *temporary*: an object holding a value over a short sequence of code,

[1352.1] Programs written in dialects of Basic that limit the number of objects that can be used (e.g., using single letters of the alphabet to denote identifiers limits the number of objects to 26) are often forced to use the same object for different purposes in different parts of the source (e.g., using it as a loop counter, then to hold the result of some calculation, then as a flag, and so on).

- *other*: all other objects.

In other, more strongly typed, languages role information is sometimes encoded as part of the type. For instance, two different integer types representing apples and oranges might be defined, with the intent that any attempt to add an object having type apple to an object having type orange will cause a diagnostic to be issued.

Is there a worthwhile benefit in limiting every object to having a single role? Experience suggests that most objects are used in a way that involves a single role (at least in programs written by non-novice developers). The storage cost associated with defining additional objects is rarely a significant consideration (although there are application domains where this cost can be very significant; and some translators attempt to minimize the storage used by a program).

1354 storage layout

Experience shows that using the same object is more than one role increases the effort needed to comprehend the code and the likelihood of faults being introduced (e.g., developers be aware of and remember the disjoint regions of source code over which an object has a particular role). The concept of an objects role is relatively new and is not specified in these coding guidelines in sufficient detail to be automatically enforced. However, this is an issue that can be covered during code review.

Rev 1352.1

If an object has one of the roles defined by these coding guidelines it shall always be used in a way that is consistent with that role.

1353 A *definition* of an identifier is a declaration for that identifier that:

definition identifier

Commentary

This defines the term *definition*. It excludes some kinds of identifiers (labels and tags), which can be declared, and macros, which are said to be *defined* but are discussed in a separate subclause. It is common developer usage to refer to a declaration of an identifier as a label or tag— as its definition. The standard also defines the terms *external definition* (which is a definition) and *tentative definition* (which eventually might cause a definition to be created). Limits on the number of definitions of an identifier are specified elsewhere.

405 label declared implicitly
1463 tag declare
1931 macro object-like
1933 macro function-like
1848 object external definition
1817 external definition
1849 tentative definition
1812 definition one external
1813 internal linkage exactly one external definition

C++

The C++ wording is phrased the opposite way around to that for C:

> *A declaration is a definition unless . . .*

The C++ Standard does not define the concept of tentative definition, which means that what are duplicate tentative definitions in C (a permitted usage) are duplicate definitions in C++ (an ill-formed usage).

3.1p2 1849 tentative definition

```
1  int glob;     /* a tentative definition */
2                // the definition
3
4  int glob = 5; /* the definition */
5                // duplicate definition
```

Other Languages

Only languages that support separate translation of source files, or some form of forward declarations, need to make a distinction between declarations and definitions.

Coding Guidelines

Many developers do not distinguish between the use of the terms *declaration* and *definition* when discussing C language. While this might be technically incorrect, in the common cases the distinction is not important. The benefit of educating developers to use correct terminology is probably not worth the cost.

object reserve storage — for an object, causes storage to be reserved for that object; 1354

Commentary

declaration 1348 syntax lifetime 451 of object

The Declarations clause of the C Standard does not specify what causes storage to be reserved for an object. This information has to be deduced from the definition of an object's lifetime.

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

3.1p2 *. . . , it contains the* **`extern`** *specifier (7.1.1) or a linkage-specification*[24] *(7.5) and neither an* `initializer` *nor a* `function-body`*, . . .*

7p6 *An object declaration, however, is also a definition unless it contains the* **`extern`** *specifier and has no initializer (3.1).*

Common Implementations

storage layout byte 54 address unique unary & 1090 operator register 1369 storage-class program 141 image operator 1000 () identifiers 286 number in block scope

Where is storage reserved for the object? The individual bytes of an object have a unique address, but unless its address is taken, storage need not be allocated for it. The issue of maintaining objects in registers only is discussed elsewhere.

In the majority of implementations storage for static duration objects is allocated in a fixed area of storage during program startup and that for automatic duration objects on a stack. Many translators assign storage to objects in the order in which they encounter their declarations during translation.

Most function definitions do not contain very many object definitions and the combined storage requirements of these definitions is not usually very large (see Figure 294.1). Many processors (including RISC) contain a form of addressing designed for efficient access to local storage locations allocated for the current function invocation— *register* + *offset* addressing mode. The processor designers choosing the maximum possible value of *offset* to be large enough to support commonly occurring amounts of locally allocated storage. In those cases where the offset of a storage location is outside of the range supported by a single instruction, multiple instructions need to be generated. The following are two opportunities for optimizing the allocation of locally defined objects to storage locations:

register 1000 + offset

1. When the quantity of local storage required by a function definition is larger than the maximum *offset* implicitly supported by the processor. Many processors designed for embedded systems applications support relatively small *offset*s. Accesses to objects with greater offsets require the use of multiple instructions, making them more costly in time and code size. Burlin[184] uses a greedy algorithm to assign storage locations to objects, based on their access costs and the amount of storage they required.

cache 0

2. On processors that contain a cache, reading a value from a storage location will also cause values from adjacent storage locations to be read (the number of storage locations read will depend on the size of a processor's cache line). Ordering objects in storage so that successive accesses to different storage locations, during program execution, are on the same cache line results in several savings.[1050] The access time to other storage locations on the same cache line is reduced and the turnover of cache lines (filling a new cache line requires the contents of an existing line to be overwritten) is reduced. The relative order of objects on a cache line may also need to be considered. On most processors the contents of a cache line are filled, from storage, in a particular order. In most cases the first location loaded is the one that caused the cache miss, followed by the next highest storage location, and so on (wrapping around to any lower addresses). For instance, assuming a 32-byte cache line filled 4 bytes at a time, a read from address xx04 that causes a cache miss, will result in storage locations xx00 through

xx1f being loaded in the order xx04, xx05, . . . , xx1f, xx00, xx01, xx02, xx03. Some processors fill the cache line completely before its contents become available, while others (e.g., IBM PowerPC) forward bytes as soon as they become available. Four successive bytes will become available on each of eight successive clock cycles. (There will be some storage latency before the first four bytes arrive.) Matching the order of objects in the cache line to the order in which they are most frequently accessed at execution-time minimizes latency. Knowing which objects are frequently accessed requires information on the execution time behavior of a program.[1186]

The assignment of objects, with static storage duration, to storage locations is usually performed by the linker. The relative ordering is often the same as the order in which the linker encounters object definitions within object files, which may or may not be the order in which the translator encounters them in the source file. The following are three main opportunities for optimizing the allocation of objects with static storage duration.

1. Some implementations use *absolute* addressing to access objects and others use *register+offset* (where the register holds the base address of storage allocated to global data). In both cases there will be a limit on the maximum offset that can be accessed using a single instruction— a larger offset requires more instructions.

   1000 register + offset

2. The cache issues are the same as for local objects.

3. Some processors (usually those targeted at embedded systems[612]) include a small amount of on-chip storage. Also some processors support pointers of different sizes (i.e., 8-bit and 16-bit representations), with the smaller size executing more quickly but only being able to access a relatively small area of storage. Deciding what objects to allocate to those storage locations that can be accessed the quickest requires whole program analysis. Sjôdin and von Platen[1250] mapped this optimal storage-allocation problem to an integer linear-programming problem and were able to achieve up to 8% reduction in execution time and 10% reduction in machine code size. Avissar, Barua, and Stewart[66] also used linear programming to obtain an 11.8% reduction in execution time, but by distributing the stack over different storage areas obtained a 44.2% reduction in execution time.

Optimizing storage layout for embedded applications can involve trading off execution time performance, amount of storage required, and power consumption.[1049] Some processors use banked storage, which can offer a potential solution to the problem of the growing difference in performance between the time taken to execute an instruction and the time taken to access a storage location. Optimizing the allocation of objects across banked storage is an active research area.[96]

559 banked storage

The total amount of physical storage available to a program never seems to be enough for some applications. In most cases developers rely on a host's support for virtual memory management. Alternative solutions are for the program to explicitly manage its own large storage requirements, or for the translator to provide a mechanism that allows developers to specify what objects should be swapped to disk and when (this usually involves the use of extensions[262]).

Many storage allocation algorithms,[1471] for allocated storage duration, are based on measurements of programs that have relatively short running times. Programs with long running times (e.g., server applications,[1138] and multithreaded applications[845]) are less well represented.

A storage allocation policy based on *best fit* (i.e., using the smallest available free storage large enough to satisfy the request) has been found to give good results in practice.[679] The `CustoMalloc`[526] tool uses dynamic profile information, on the size of the object requests made, to build a custom storage allocator tuned to a given program (or at least the profiling data). Execution time profiling of references to objects with allocated storage duration can also be used to improve a program's performance by segregating objects with similar behaviors (leading to improved reference locality).[1211]

Various other storage layout issues are discussed in other C sentences.

0 cache
39 alignment
530 structure type
sequentially allocated objects
994 array

Coding Guidelines

For a variety of reasons some developers take advantage of the relative addresses of storage allocated to objects. There is no guarantee where block scope objects will be allocated storage relative to each other, and the relative addresses of objects having static storage duration is even more unpredictable. There is a simpler method of outlawing such usage than enumerating all cases where a program could depend on the layout of block scope storage. The following guideline renders any such dependence useless.

Cg 1354.1 Arbitrarily reordering the object declarations in the same scope shall not result in a change of program output.

Dev 1354.1 When an object declaration contains a dependency on another object declared in the same scope the arbitrary reordering may be restricted to those orderings that do not result in a constraint violation.

Example

```
1  #include <string.h>
2
3  extern int e_g_1;  extern int e_g_2;
4  static int s_g_1;  static int s_g_2;
5
6  void f(void)
7  {
8         int loc_1;         int loc_2;
9
10 if ((&e_g_1 + 1) == &e_g_2)
11    {
12    memset(&e_g_1, sizeof(e_g_1) * 2, 0xff);
13    /*
14     * No requirement that (&s_g_1 + 1) == &s_g_2) be true.
15     */
16    memset(&s_g_1, sizeof(e_g_1) * 2, 0xff);
17    /*
18     * No requirement that (&loc_1 + 1) == &loc_2) be true.
19     */
20    memset(&loc_1, sizeof(e_g_1) * 2, 0xff);
21    }
22 }
```

— for a function, includes the function body;[99)] 1355

Commentary

function definition 1821 syntax

A function definition is intended to contain executable statements (that may perform some action) which occupy storage space.

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

3.1p2 *A declaration is a definition unless it declares a function without specifying the function's body (8.4), . . .*

Other Languages

Languages that have a mechanism to support the building of programs from separately translated source files usually refer to the function declaration that contains its body as its definition.

1356 — for an enumeration constant or typedef name, is the (only) declaration of the identifier.

Commentary

Since only one declaration is allowed in the same scope for these kinds of identifiers it might be more accurate to always call it a definition. However, using this convention would create more wording in the standard than it saves (because some of the wording that applies to declarations would then need to be extended to include definitions; specifying these identifiers to be both declarations and definitions removes the need for this additional wording).

C90

The C90 Standard did not specify that the declaration of these kinds of identifiers was also a definition, although wording in other parts of the document treated them as such. The C99 document corrected this defect (no formal DR exists). All existing C90 implementations known to your author treat these identifiers as definitions; consequently, no difference is specified here.

C++

A declaration is a definition unless . . . , or it is a **`typedef`** *declaration (7.1.3), . . .* 3.1p2

A typedef name is not considered to be a definition in C++. However, this difference does not cause any C compatibility consequences. Enumerations constants are not explicitly excluded in the *unless* list. They are thus definitions.

Other Languages

Languages that contain these constructs usually refer to their declarations as definitions of those identifiers.

1357 The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote.

declaration specifiers

Commentary

This says in words what the syntax specifies. Like the syntax there are no restrictions placed on the ordering of specifiers.

C++

The C++ Standard does not make this observation.

Common Implementations

Some implementations contain extensions that add to the list of attributes that can be specified by sequences of specifiers. These are dealt with in their respective sentences.

Coding Guidelines

Are there any advantages to imposing some kind of ordering on declaration specifiers? There would be if all developers used the same ordering (on the basis that source code readers would not have to scan a complete declaration looking for the information they wanted; like looking a word up in a dictionary, it would be in an easy-to-deduce position). Analysis of existing source (see Table 1364.1, Table 1378.1, and Table 1476.1) shows that the most commonly seen ordering is *`storage-class-specifier type-qualifier type-specifier`*. Given developers' existing experience with this ordering, the likelihood that it will be the ordering they are most likely to encounter in the future, and the lack of any obvious benefit to using another ordering, the recommendation is to continue using this ordering.

Cg 1357.1 Within a declaration the declaration specifiers shall occur in the order *`storage-class-specifier type-qualifier type-specifier`*.

storage-class specifiers 2039
future language directions
type specifiers 1381
ordering

Placing a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is specified as being obsolescent (the same specification also appeared in C90). The relative ordering of *type-specifier* is discussed elsewhere.

Example

```
int const typedef I;
double const long static J;
int volatile long extern long const unsigned K;
```

---

declarator list of

The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. 1358

Commentary

This says in words what is specified in the syntax.

C++

The C++ Standard does not make this observation.

Coding Guidelines

The C declaration syntax allows type information to be specified in the *declarator*. It is possible for different *init-declarator*'s, each declaring a different identifier, in an *init-declarator-list* to declare identifiers having different types.

```
int i_1,
    *p_1,        /* *p_1 is the declarator, a pointer to ... */
    **p_2,       /* **p_2 is the declarator, a pointer to pointer to ... */
    * const p_3, /* * const p_3 is the declarator, a const pointer to ... */
    a_1[10],     /* a_1[10] is the declarator, an array of ... */
    *(a_p_1[4]); /* *(a_p_1[4]) is the declarator, an array of pointer to ... */

char *pc_1,
      pc_2;
```

A mistake commonly made by inexperienced developers is to assume that pc_2, in the above example, has type pointer to **char**. A similar mistake does not seem to occur very often for array types. This may be due to the significantly smaller number of array declarations, compared to pointer declarations, in C or because the additional tokens are not visually contiguous with the declaration specifiers but to the right of the identifier being declared.

Cg 1358.1

A *declarator* specifying a pointer type shall not occur in the same *init-declarator-list* as *declarator*s not specifying a pointer type or specifying a different pointer type.

Objects that are used to store the same set of values are usually declared to have the same type. A change to the set of values that needs to be represented usually requires changing the declarations of all the associated objects. The following are a number of techniques that can be used to declare more than one object:

- Multiple *declaration*s each of whose *init-declarator-list* contains a single *init-declarator*.

```
int i;
int j;
int k;
```

Some coding guideline documents recommend this technique. The rationale is that it reduces the likelihood of cut-and-paste editor operation mistakes, or modifications to the *declaration-specifiers* having unexpected results. (Because a single object is declared, it is not possible to affect the declaration of another object.) The disadvantage of this approach is that, if it is necessary to change the *declaration-specifiers* associated with a set of object declarations, each *declaration* has to be modified (it is possible this change will not be applied to some of them).

- A single *declaration* whose *init-declarator-list* contains more than one *init-declarator*.

```
1  int i,
2      j,
3      k;
```

The advantages and disadvantages of this technique are those of the above, reversed. Experienced developers will be familiar with cut-and-paste mistakes when modifying this kind of declaration. The extent to which either form of declaration forms a stronger visual association, or reduces the effort needed to read the identifier list, is not known.

- Multiple *declaration*s using a typedef name.

```
1  typedef int francs; /* French francs */
2
3  francs i;
4  francs j;
5  francs k;
```

This approach appears to solve the *declaration-specifiers* modification disadvantages associated with the first technique and yet has its advantages, but only because it associates a unique name, `francs`, with the type that is common to the three declared objects.

822 symbolic name

- Some combination of the above.

Without any experimental evidence, or record of faults introduced, it is not possible to verify any claims about either of the first two techniques being more or less prone to the creation of faults.

Some coding guideline documents recommend that *type-specifier*s other than *typedef-name* only appear in the definition of a typedef name. Such a guideline recommendation is easily followed without addressing the underlying issues. For instance, in the `francs` example above, using a typedef name of `INT` does not solve any of the problems. Because the name `INT` is not specific to the information represented in `i`, `j`, and `k` only. Such a typedef name is also likely to be used to declare objects that do not hold values denoting French francs. This issue is discussed in more detail elsewhere.

1629 typedef name syntax

1359 99) Function definitions have a different syntax, described in 6.9.1.

footnote 99

Commentary

During syntactic analysis of a sequence of tokens, the first difference between a function declaration and a function definition occurs when either a **;** or **{** token is encountered (which in the case of **;** is the last token of a function declaration).

1821 function definition syntax

Other Languages

In most languages the syntax for function definitions is usually very different from object definitions. The evolving designs of object-oriented languages is starting to blur the distinction.

1360 The declarators contain the identifiers (if any) being declared.

Commentary

A declarator always declares at least one identifier. However, a declaration need not include a declarator, although it must declare some identifier. An abstract declarator need not declare any identifiers.

1349 declaration shall declare identifier
1624 abstract declarator syntax

object type complete by end

If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; 1361

Commentary

incomplete types 475
footnote 109 1465

The rationale behind the support of incomplete types does not apply to objects having no linkage. Objects with other kinds of linkage may be declared to have an incomplete type. An initializer can only complete an incomplete array type in this context. Requirements on the completion of types of objects having external and internal linkage are discussed elsewhere.

object type for internal linkage 1851

C++

3.1p6 *A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).*

The C++ wording covers all of the cases covered by the C specification above.

A violation of this requirement must be diagnosed by a conforming C++ translator. There is no such requirement on a C translator. However, it is very unlikely that a C implementation will not issue a diagnostic in this case (perhaps because of some extension being available).

Other Languages

Those languages supporting declarations that need storage allocation decisions to be made at translation time (in nearly all cases this applies to locally declared objects) have similar requirements.

Common Implementations

Although not required to do so, all known implementations issue a diagnostic if this requirement is not met.

Coding Guidelines

array of unknown size initialized 1683

The issues associated with using the number of initializers to specify the number of elements in an array type are the same for declarations having any linkage, and are discussed elsewhere.

---

in the case of function ~~arguments~~ parameters (including in prototypes), it is the adjusted type (see 6.7.5.3) that is required to be complete. 1362

Commentary

array converted to pointer 729

parameter adjustment in definition 1595

Any adjusted parameter type (the case applies to function parameters, not arguments; this is a typo in the standard) will have converted incomplete array types into pointers to their element types (there are no incomplete, or complete, function types and their adjusted type is not relevant here). No other kind of incomplete types can be completed by such adjustments. The constraint in clause 6.7.5.3 applies to function definitions.

The wording was changed by the response to DR #295.

C90

This wording was added to the C99 Standard to clarify possible ambiguities in the order in which requirements, in the standard, were performed on parameters that were part of a function declaration; for instance, `int f(int a[]);`.

C++

The nearest the C++ Standard comes to specifying such a rule is:

5.2.2p4 *When a function is called, the parameters that have object type shall have completely-defined object type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type.]*

Other Languages

Most other languages do not treat the type of function arguments any different from other object types.

Example

```
1  extern int f_1(int p_1[]);
2  extern int f_2(int p_1[4]);     /* Number of elements is not considered. */
3
4  extern int f_3(struct foo p_1); /* foo cannot be completed. */
```

1363 **Forward references:** declarators (6.7.5), enumeration specifiers (6.7.2.2), initialization (6.7.8).

6.7.1 Storage-class specifiers

1364

storage-class specifier syntax

storage-class-specifier:
typedef
extern
static
auto
register

Commentary

The character sequence *regist* is not generated by the algorithms commonly used by speakers of English for abbreviating *register*.

792 abbreviating identifier

C++

The C++ Standard classifies **typedef** (7.1p1) as a *decl-specifier*, not a *storage-class-specifier* (which also includes **mutable**, a C++ specific keyword).

Other Languages

Languages often use the keyword **type** to denote that a type is being declared, although a few languages (e.g., Algol 68 and CHILL) use **mode**, or some variation of that word. Java is unusual, in a modern language, in not providing a mechanism for defining a name to have a primitive (scalar in C terminology) type or array type.

Some languages also use the keyword **extern**. Fortran uses the keyword **extern** to declare a parameter as denoting a callable function (it does not have function types as such).

Some languages (e.g., CHILL) provide a mechanism for specifying which registers are to be used to hold objects (CHILL limits this to the arguments and return value of functions). The keyword **register** is unique to C (and C++).

Pascal requires that the keyword **forward** on procedure and function declarations that are defined later in the same source file.

Coding Guidelines

The standard only uses the keyword **auto** in a few places (outside of comments in examples). However, the phrase *automatic storage duration* occurs much more often. An occurrence of the keyword **auto** in the visible source provide little additional information to a reader. A declaration's lexical position with respect to being inside/outside of a function definition, or the presence of other storage-class specifiers provides all the information required by a reader. As the Usage figures show (see Table 1364.1) existing practice is not to use this keyword. A guideline recommending against its use would be redundant.

1372 footnote 100
1765 for statement declaration part
1811 external declaration not auto/register
457 automatic storage duration

Table 1364.1: Common token pairs involving a *storage-class*. Based on the visible form of the `.c` files (the keyword **auto** occurred 14 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
static void	33.7	32.7	**extern int**	32.1	1.7
static int	28.2	15.1	**register struct**	19.1	1.4
typedef union	3.2	11.0	**typedef struct**	62.4	1.2
static const	1.5	10.0	**register int**	23.0	1.2
static volatile	0.3	8.6	**register char**	10.2	1.2
typedef enum	10.8	8.2	**register unsigned**	6.1	0.9
static signed	0.0	6.5	**extern char**	7.4	0.9
static unsigned	3.8	5.5	**extern struct**	6.9	0.5
extern double	1.3	5.5	**static** identifier	21.0	0.3
static char	4.1	5.1	**typedef unsigned**	6.2	0.2
static struct	6.4	4.8	**typedef** identifier	7.9	0.0
register enum	1.6	4.6	**register** identifier	35.9	0.0
extern void	21.5	2.1	**extern** identifier	23.7	0.0

Table 1364.2: Common token pairs involving a *storage-class*. Based on the visible form of the `.h` files (the keyword **auto** occurred 6 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
typedef union	12.4	67.1	**typedef unsigned**	6.6	3.1
typedef enum	6.2	37.2	**extern unsigned**	2.9	2.8
typedef signed	0.5	28.6	**static void**	10.3	2.2
extern void	28.6	24.0	**typedef void**	4.0	1.6
extern double	0.3	17.9	**static int**	7.0	1.2
typedef struct	46.3	16.6	**extern** identifier	32.2	0.9
extern int	23.2	15.2	**register long**	16.0	0.8
extern float	0.3	9.8	**register unsigned**	24.8	0.6
register signed	2.6	8.2	**static** identifier	70.3	0.5
static const	6.4	5.0	**register int**	18.4	0.3
extern char	3.8	4.8	**typedef** identifier	16.7	0.2
extern struct	4.3	3.3	**register** identifier	18.4	0.0

Constraints

At most, one storage-class specifier may be given in the declaration specifiers in a declaration.[100)] 1365

Commentary

storage 448
duration
object
linkage 420

The storage-class specifier can be used in a declaration to specify two attributes: the storage duration, and linkage. There is only one context where the appearance of a storage-class specifier in a declaration can affect the storage duration of the object being declared. An object declared in block scope has automatic storage duration unless either of the keywords **extern** or **static** appear in its declaration. In which case it has static storage duration. The presence of the storage-class specifiers **typedef**, **extern**, and **static** may cause the default linkage given to an identifier, because of where it is declared in the source, to be changed.

C++

While the C++ Standard (7.1.1p1) contains the same requirement, it does not include **typedef** in the list of *storage-class-specifiers*. There is no wording in the C++ limiting the number of instances of the **typedef** *decl-specifier* in a declaration.

Source developed using a C++ translator may contain more than one occurrence of the **typedef** *decl-specifier* in a declaration.

Other Languages

In many languages the location of an objects declaration in the source code is used to specify its storage class. Like C, some languages provide keywords that enable the default storage class to be overridden, e.g., Fortran **common**.

Coding Guidelines

Should a storage-class specifier ever appear in a declaration? The only two that are ever necessary are **typedef** and **static** (to change default behavior; **extern** in block scope is not necessary because the declaration can be moved to file scope). All other uses are related to coding guidelines issues. The use of **extern** is covered by the guideline recommendation dealing with a single point of declaration.

422.1 identifier declared in one file

The storage duration and linkage issues associated with the storage-class specifier **static** are discussed elsewhere. The efficiency issues associated with the storage-class specifier **register** are discussed elsewhere.

425 static internal linkage
455 static storage duration
1369 register storage-class

Semantics

1366 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only;

Commentary

The keyword **typedef** is not really a storage class as such. However, the syntax for typedef name declarations is the same as that for object and function declarations. The committee considered it a worthwhile simplification to treat **typedef** as a storage class.

C++

It is called a *decl-specifier* in the C++ Standard (7.1p1).

Other Languages

In other languages the keyword used to define a new type is rarely placed in the same syntactic category as the storage-class specifiers.

1367 it is discussed in 6.7.7.

1368 The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

1369 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible.

register storage-class

Commentary

It is the developer who is making the *suggest*ion to the translator (sometimes the term *hint*) is used). The original intent of this storage-class specifier was to reduce the amount of work a translator vendor needed to do when implementing a translators machine code generator (which also contributed to keeping the overall complexity of a C translator down). This suggestion is often interpreted by developers to mean that translators will attempt to keep the values of objects declared using it in registers. Had C been designed in the 1990s the keyword chosen might have been **cache**.

0 cache

The standard does not permit declarations at file scope to include the storage-class specifier **register**.

1811 external declaration not auto/register

C++

> *A* ***register*** *specifier has the same semantics as an* ***auto*** *specifier together with a hint to the implementation that the object so declared will be heavily used.* 7.1.1p3

Translator implementors are likely to assume that the reason a developer provides this hint is that they are expecting the translator to make use of it to improve the performance of the generated machine code. The

C++ hint does not specify implementation details. The differing interpretations given, by the two standards, for hints provides to translators is not likely to be significant. The majority of modern translators ignore the hint and do what they think is best.

Other Languages

While it might be possible to make use of particular techniques (e.g., defining the most frequently accessed objects first) to improve the quality of machine code generated by translators of any language, the specification of a language's semantics rarely includes such hint mechanisms.

Common Implementations

Processors rarely perform operations directly on values held in storage, they move them to temporary working locations first. In nearly all cases these temporary working locations are a small set of storage areas, known as *registers*, in the processor itself. Some processors have an architecture that is stack-based[762] rather than register-based and the Bell Labs C Machine[363] was a stack-based processor specifically designed to execute C. However, experience with the two architectural choices has shown that the register-based approach yields better overall performance and the majority of modern processes use it. Stack-based processors are not as common as they once were, although new ones are still occasionally built.

expression 940 processor evaluation

What is a register? The documentation for the Ubicom IP2022[1384] says it has 255 registers. However, the transistors used to represent the values of these registers (at least in the current implementation) are in the same area of the chip as the rest of storage. In fact these *registers* occupy the lower portion of the processors address space. They are accessed using a direct addressing mode (the same could be said about registers in other processors, except that rarely share an address with the rest of storage).

Customer demand for higher-performance (in the generated machine code) and competition between translator vendors means that vendors desire to minimize the cost of producing a code generator no longer includes not implementing a sophisticated register allocator.

Using the storage-class specifier **`register`** to help improve the performance of the generated machine code can be a difficult process that sometimes has the reverse affect (programs that execute more slowly). Translator implementors are aware that developers expect objects declared using **`register`** to have their values kept in registers when possible. Some implementors have decided to attempt to meet this expectation, for the duration of the objects lifetime. The result can be a decrease in performance, because a register is used to hold a particular objects value during a sequence of statements where it would have been better for that register to be holding a different objects value. Other translator implementors believe that the known register allocation algorithms produce better results and ignore any developer provided hints.

The values of objects are not the only things that may be worth trying to keep in registers. If a sequence of code performs the same operation on two operands, whose values have not changed between the two occurrences (a common subexpression), keeping the result in a register and reusing it may be more efficient than performing the operation again. Use of the **`register`** storage-class specifier can have an indirect benefit. An optimizer does not need flow analysis to deduce that the object is not aliased; it is not permitted to as the operand of the address-of operator.

common 1712 subexpression

unary & 1088 operand constraints

register optimizing allocation

A number of algorithms have been proposed for allocating values to registers (it is known that optimal register allocation is NP-hard[408]). An equivalence between this problem and the pure mathematics problem of graph coloring has been shown to exist.[159] So called *register coloring* algorithms have proved to be very popular for one class of processors (those having many registers that are treated orthogonally) and a variety of different variations on this approach have been used. When the time taken to translate code is important (e.g., just in time compilation) *linear scan* register allocation[1099] is much faster than register coloring and has been found to result in code that executes only 12% slower than an aggressive register allocator.

expression 940 optimal evaluation

It has proved difficult to find general algorithms for register allocation on processors having few registers, or where there are restrictions on the operations that can be performed using some registers. In many cases the allocation algorithms are hand tailored for each processor. Mapping the problem to one in integer linear programming has produced worthwhile results for processors having few registers, such as the Intel Pentium.[51]

array element held in register

Register allocation algorithms generally only consider trying to maintain objects having a scalar type in registers. A study by Li, Gu, and Lee[854] evaluated the benefits of doing the same for array elements, known as *scalar replacement* (see Chapter 8 of Allen[17]). They used trace driven simulations of a variety of Fortran benchmark programs to obtain idealised (tracing the actual data references) measurements to obtain the best results that could be achieved, if an optimizer optimally assigned objects to registers. The results showed that having between 48 and 96 registers available, for holding scalar and array element values, had the most impact (savings were even possible with 32 available registers). On five C based multimedia kernels an implementation of scalar replacement by So and Hall[1262] reduced memory accessed by 58% to 90% and saw speedups of between 2.34 and 7.31.

A number of studies have investigated the affects of increasing or decreasing the number of registers available to a translator on the performance of the generated machine code. Having sufficient registers to be able to keep all object values in one of them may be the ideal. In practice optimizers are limited by the analysis they perform on the source. For instance, if it is not possible to deduce whether a particular object is referenced via a pointer, an optimizer has to play safe and access the value from storage. Performance improvements have been found to more or less peak at 17 registers using lcc,[112] later analysis using more sophisticated optimizations (including inlining and allocating globals to registers) found that effective use could be made of 64 registers.[1110]

Many early processors tended to have few registers because of hardware complexity and cost considerations. The relative importance of these considerations has decreased over time and having 32 registers is a common theme among modern processors. However, in practice once function calling conventions are taken into account and various registers reserved (for a variety of reasons, for instance holding the stack pointer) there are rarely more than 16, out of 32, truly temporary registers available to a translator.

485 ABI
1811 external declaration not auto/register

Some implementations support the use of the **`register`** storage-class specifier on declarations at file scope. Such declarations are usually specifications of which register should be dedicated to holding a particular object.[55, 623]

Franklin and Sohi[442] measured register usage while programs from the SPEC89 benchmark executed on a MIPS R2000 (see Table 1369.1).

Table 1369.1: Degree of use of floating-point and integer register instances (a particular value loaded into a register). Values denote the percentage of register instances with a particular degree of use (listed across the top), for the program listed on the left. For instance, 15.51% of the integer values loaded into a register, in `gcc`, are used twice. Left half of table refers to floating-point register instances, right half of table to integer register instances. Zero uses of a value loaded into a register occur in situations such as an argument passed to a function that is never accessed. Adapted from Franklin and Sohi.[442]

Usage	0	1	2	3	≥4	Average	0	1	2	3	≥4	Average
eqntott							0.89	71.34	17.54	9.47	0.76	1.86
espresso							3.67	72.30	17.66	3.74	2.63	1.48
gcc							6.26	67.37	15.51	4.45	6.41	1.69
xlisp							4.27	66.14	12.42	10.20	6.97	1.84
dnasa7	0.00	99.83	0.02	0.03	0.12	1.31	0.67	2.36	16.29	64.36	16.33	3.28
doduc	1.46	84.00	9.51	1.94	3.09	1.36	10.31	44.35	26.52	10.13	8.69	2.93
fpppp	0.16	91.09	6.15	1.14	1.46	1.16	1.34	10.12	83.45	0.46	4.63	3.09
matrix300	0.00	99.92	0.00	0.00	0.08	1.25	15.29	61.54	7.71	0.12	15.35	1.92
spice2g6	0.21	79.85	19.22	0.16	0.56	1.22	4.04	73.38	12.08	3.56	6.94	1.68
tomcatv	0.00	86.43	8.30	1.49	3.77	1.26	0.12	24.99	37.54	27.40	9.96	3.22

1370 The extent to which such suggestions are effective is implementation-defined.[101)]

register extent effective

Commentary

All translators need to have a register allocation algorithm to be able to generate executable machine code. The extent to which the **`register`** storage-class specifier affects the behavior of this algorithm needs to be documented. A translator's documented behavior is unlikely to be sufficient to enable a developer to predict

which values will be held in which processor register. The possible permutations are rarely sufficiently small that the behavior is easily enumerated.

C++

The C++ Standard gives no status to a translator's implementation of this hint (suggestion). A C++ translator is not required to document its handling of the **register** storage-class specifier and often a developer is no less wiser than if it is documented.

Coding Guidelines

Those coding guideline documents that base their recommendations on the list given in annex I are likely to recommend against the use of the **register** storage-class specifier. The only externally visible affect of using the **register** storage-class specifier is a possible change in execution time performance or size of program image. In both cases the changes are unlikely to be worth a guideline.

program image 141

Example

Macros provide a flexible method of controlling the definitions that contain the **register** storage-class specifier.

```
#if EIGHT_BIT_CPU != 0
#define REG1 register
#define REG2
#define REG3
#define REG4
#endif

#if MODERN_DSP != 0
#define REG1 register
#define REG2 register
#define REG3
#define REG4
#endif

#if RISC_CHIP != 0
#define REG1 register
#define REG2 register
#define REG3 register
#define REG4 register
#endif

void f(void)
{
REG1 int total_valu;
REG2 short intermediate_valu;
REG3 int the_valu;
REG4 long less_often_used_value;

/* ... */
}
```

---

block scope storage-class use

The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**. 1371

Commentary

There is a lot of existing source containing function declarations at block scope using the **extern** storage-class specifier. The Committee did not want to render such usage as undefined behavior and decided to permit it.

Other Languages

Most languages do not support the declarations of identifiers for functions in block scope. Although some languages do support nested function definitions.

Common Implementations

Translators usually flag an occurrence of this undefined behavior.

Coding Guidelines

If the guideline recommendation dealing with function declarations at file scope is followed this requirement is not an issue.

422.1 identifier declared in one file

1372 100) See “future language directions” (6.11.5).

footnote 100

1373 101) The implementation may treat any **register** declaration simply as an **auto** declaration.

footnote 101

Commentary

That is to say, an implementation may treat such a declaration as an **auto** declaration for the purposes of storage allocation. However, the various constraints and other kinds of behavior associated with an object declared using the **register** storage class still apply.

1088 unary & operand constraints

1374 However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary **&** operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1).

Commentary

A constraint violation occurs if the operand of the address-of operator has been declared using storage-class specifier **register**.

1088 unary & operand constraints

This observation applies when the type category is an array type. But, it does not apply to the case where a member of a structure or union type has an array type. For instance:

553 type category

```
1  struct {
2          register int mem[100];
3          } x;
4  &x.mem;     /* &     applied to an array type. */
5  &x;         /* & not applied to an array type. */
```

C++

This requirement does not apply in C++.

1088 unary & operand constraints

1375 Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

Commentary

An operand having an array type is not converted to pointer type when it is operated on by the address-of or **sizeof** operator. The former would be a constraint violation, leaving the latter.

729 array converted to pointer
1088 unary & operand constraints

C++

This observation is not true in C++.

1088 unary & operand constraints

1376 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

Commentary

member 433
no linkage
storage 448
duration
object

Members of a structure or union type always have no linkage.

Another property resulting from the presence of a storage-class specifier is the storage duration of an object. The other properties might more properly be called *consequences*. The consequences arising from wording in other parts of the Standard, such as Constraints, undefined and implementation-defined behaviors. For instance, while placing the **register** storage-class specifier on the declaration of an object having a structure type may not result in any of its members being held in processor registers, the associated constraints still apply (it is a constraint violation for the result of a member selection operator to appear as the operand of the address-of operator).

register 1369
storage-class

C90

This wording did not appear in the C90 Standard and was added by the response to DR #017q6.

C++

The C++ Standard does not explicitly specify the behavior in this case.

Other Languages

In many object-oriented languages it is possible to specify that members of classes have a different storage-class, or linkage, than the class itself.

Forward references: type definitions (6.7.7). 1377

6.7.2 Type specifiers

type specifier
syntax

1378

type-specifier:
void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
~~**_Imaginary**~~
struct-or-union-specifier
enum-specifier
typedef-name

Commentary

declaration 1348
syntax

The syntax of *type-specifier* does not enumerate all possible combinations of keywords that may be used to specify an arithmetic type. The syntax for *declaration-specifiers* supports the occurrences of more than one *type-specifier* in a single declaration.

The wording was changed by the response to DR #207.

C90

Support for the *type-specifiers* **_Bool**, **_Complex**, and **_Imaginary** is new in C99.

C++

The nonterminal for these terminals is called *simple-type-specifier* in C++ (7.1.5.2p1). The C++ Standard does contain a nonterminal called *type-specifier*. It is used in a higher-level production (7.1.5p1) that includes *cv-qualifier*.

The C++ Standard includes **wchar_t** and **bool** (the identifier bool is defined as a macro in the header **stdbool.h** in C) as *type-specifiers* (they are keywords in C++). The C++ Standard does not include **_Bool**, **_Complex** and **_Imaginary**, either as keywords or type specifiers.

Other Languages

Equivalent *type-specifiers* seen in other languages include: **integer**, **short real**, **real**, **double precision**, **boolean**, **character**, **ptr**, and **complex**.

Common Implementations

Implementations add additional type specifiers, as extensions, either to support some special purpose hardware functionality (e.g., the Intel 8051 processor[625] has an area of storage containing what are known as *special function registers*). Some vendors (e.g., Keil,[717] Tasking[20]) have added type specifiers (e.g., **sfr** and **sbit**) to provide a mechanism for declaring objects denoting locations in this storage area, or to give special semantics to objects that are application specific.

Motorola added some vector functionality (what it calls AltiVec Technology Resources[971]) to its implementation (the MPC7400) of the IBM PowerPC instruction set. This enabled 128 bits of storage to be treated as either: sixteen 8-bit objects, eight 16 bit objects, or four 32-bit objects. Their translator for this processor supports the keyword **__vector** as an extension. This keyword acts as a type specifier, indicating that for instance an object is composed of 16 unsigned chars. Operations on and between objects declared with this type specifier operate on all of the subobjects at the same time. For instance, an add operation will add corresponding unsigned chars from each of the two operands. The streaming SIMD extensions (SSE)[627] to the Intel x86 instruction set support a similar set of operations (see Figure 1378.1) on blocks of 128 bits and a few translators[630] make use of them.

Coding Guidelines

A guideline recommendation dealing with the use of a subset of the available type specifiers is discussed elsewhere.

480.1 object
int type only

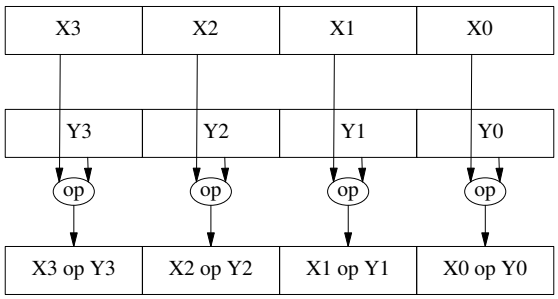


Figure 1378.1: Behavior of packed single-precision floating-point operations supported by the Intel Pentium processor.[627]

Table 1378.1: Common token pairs involving a *type-specifier*. Based on the visible form of the `.c` files. The type specifiers **_Bool**, **_Complex**, and **_Imaginary** did not appear in the visible form of the `.c` files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
unsigned long	38.7	72.2	**; long**	0.1	6.2
unsigned short	5.8	63.8	, **void**	0.3	5.8
char *p	74.5	63.3	**static unsigned**	3.8	5.5
(signed	0.0	60.5	**extern double**	1.3	5.5
; enum	0.1	45.5	**} int**	2.2	5.3
(struct	2.9	41.8	**{ signed**	0.0	5.2
; float	0.1	40.0	**static char**	4.1	5.1
; union	0.0	33.7	header-name **double**	0.2	5.1
static void	33.7	32.7	**static struct**	6.4	4.8
(float	0.0	32.0	**register enum**	1.6	4.6
(unsigned	1.0	29.0	**long *p**	7.1	2.8
(void	1.4	26.6	**int** identifier	87.6	2.3
; unsigned	1.0	26.4	**extern void**	21.5	2.1
; int	2.5	24.8	**struct** identifier	99.0	1.9
(char	1.0	23.9	**extern int**	32.1	1.7
{ union	0.0	23.4	**short *p**	21.8	1.4
(double	0.0	22.9	**register struct**	19.1	1.4
; double	0.0	19.8	**const unsigned**	6.2	1.4
void *p	17.5	19.0	**const struct**	11.1	1.3
, **unsigned**	0.6	18.9	**typedef struct**	62.4	1.2
} void	4.1	18.0	**register int**	23.0	1.2
unsigned char	21.2	18.0	**register char**	10.2	1.2
; struct	1.3	17.6	**volatile unsigned**	25.6	1.1
; char	0.8	17.5	**void** identifier	61.7	0.9
, **int**	1.4	15.9	**void)**	17.5	0.9
static int	28.2	15.1	**register unsigned**	6.1	0.9
; signed	0.0	14.7	**extern char**	7.4	0.9
{ struct	4.3	14.5	**const void**	5.3	0.8
identifier **double**	0.0	13.1	**signed short**	11.3	0.7
{ unsigned	1.9	12.5	**int)**	6.6	0.6
, **struct**	0.8	12.2	**extern struct**	6.9	0.5
{ int	4.8	11.5	**volatile struct**	15.5	0.4
{ enum	0.1	11.1	**long** identifier	68.3	0.4
typedef union	3.2	11.0	**long)**	21.7	0.4
(short	0.0	11.0	**float *p**	9.2	0.3
; short	0.0	10.6	**char** identifier	22.6	0.3
(int	1.0	10.6	**typedef unsigned**	6.2	0.2
, **float**	0.0	10.6	**signed long**	20.8	0.2
const char	54.1	10.4	**double *p**	7.9	0.2
{ float	0.1	10.2	**volatile int**	7.4	0.1
(union	0.0	9.9	**unsigned** identifier	7.0	0.1
, **char**	0.4	9.9	**union {**	34.5	0.1
(long	0.2	9.2	**signed char**	22.6	0.1
, **enum**	0.0	9.2	**short** identifier	60.9	0.1
unsigned int	24.6	9.1	**enum {**	13.4	0.1
{ double	0.0	8.6	**union** identifier	65.5	0.0
typedef enum	10.8	8.2	**signed int**	7.5	0.0
, **double**	0.0	8.2	**signed)**	37.9	0.0
int *p	4.1	8.1	**short)**	14.0	0.0
, **union**	0.0	8.0	**float** identifier	64.3	0.0
, **signed**	0.0	7.9	**float)**	26.1	0.0
) enum	0.0	7.1	**enum** identifier	86.6	0.0
{ char	1.3	7.1	**double** identifier	70.7	0.0
static signed	0.0	6.5	**double)**	19.1	0.0
; void	0.3	6.3			

Constraints

1379 At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name.

declaration at least one type specifier

Commentary

C no longer supports any form of implicit declaration.

A *specifier-qualifier-list* is the production used in the syntax for structure and union specifiers. It denotes an optional list of *type-specifier* and *type-qualifier*.

1390 struct/union syntax

C90

This requirement is new in C99.

In C90 an omitted *type-specifier* implied the type specifier **int**. Translating a source file that contains such a declaration will cause a diagnostic to be issued and are no longer considered conforming programs.

C++

At least one type-specifier *that is not a* cv-qualifier *is required in a declaration unless it declares a constructor, destructor or conversion function.*[80)]

7.1.5p2

Although the terms used have different definitions in C/C++, the result is the same.

Other Languages

Most languages require that declarations contain some form of type specification (although some of them will create an implicit declaration if an identifier is referenced without first being declared). Some languages have no means of explicitly declaring the type of an object. For instance, ML implementations need to deduce the type of an object from the context in which it is used. Fortran uses the first character of an identifier to implicitly give it a type, if it is not given one explicitly through a declaration. Identifiers starting with any of the letters `I` through `N`, inclusive, having integer type and all other identifiers having real type. There is even an **IMPLICIT** statement that allows the developer to control the types implicitly chosen for identifiers starting with different alphabetic letters.

Common Implementations

Most C90 implementations do not issue a diagnostic for the violation of this C99 constraint. It is expected that most C99 translators will continue to treat such declaration as implying the type **int**.

Coding Guidelines

Many developers will continue to use C90 translators for years to come and these will not check for a violation of this constraint. Some static analysis tools do not check source for violations of constraints on the basis that this check is performed by translators (thus reducing the amount of work that implementors of such tools need to do). The existing C90 behavior is well defined, experienced developers will be familiar with it, and it is simple for developers new to C to learn. Occurrences of an omitted type specifier in existing code is rare. Given this rarity and the small cost paid by readers of the source (often a slight confusion causing a task switch followed by the realization that the **int** *type-specifier* has been omitted).

0 cognitive switch

1380 Each list of type specifiers shall be one of the following sets (delimited by commas, when there is more than one set on a line);

type specifiers sets of

Commentary

A set is an unordered collection of items (tokens in this case). The syntax permits an arbitrary long sequence of different type specifiers. This constraint delimits the set of possible type specifier combinations that can appear in a conforming program.

Other Languages

Few languages support the large number of different combinations of type specifiers available in C. Algol 68 was unusual in that its use of a two level grammar permitted families of an infinite number of types, e.g., **int**, **short int**, **short short int**, **short short short int**, . . . , and **int**, **long int**, **long long int**, **long long long int**, and so on (implementations were allowed to specify a *shortest* and *longest* type, after which additional **short** and **long** had no affect on representation).

Coding Guidelines

If developers followed the principle of least effort, when typing source, we would expect the type specifiers most often used to be the first one listed on each line and the ones least often used to be the last on the line. With one exception existing code appears to overwhelmingly use the shortest form (see Table 1378.1). The exception is the type **unsigned int**. Developers appear to use the **unsigned** type specifier for all unsigned types. While developers will be most practiced using the shorter forms, the additional cost of using any of the longer forms is likely to be small. For this reason a guideline recommendation would not appear to have a worthwhile cost/benefit.

---

type specifiers ordering

the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers. 1381

Future language directions

Commentary

There is no future language direction specifying any change in this behavior.

Other Languages

Most languages require a fixed order for type specifiers. Even though they may contain of more than one token, they are usually treated, by the syntax, as if they were a single token. Very few languages allow other declaration specifiers to be intermixed with type specifiers.

Coding Guidelines

Existing code rarely contains a list of type specifiers having an order that is not one of those specified in the C Standard (see Table 1378.1). Based on this common practice in existing code the possibility that readers will only read what they consider to be the minimum number of tokens needed to deduce an objects integer type has to be considered. For instance, readers may believe that the declaration `long unsigned id_name` declares an object to have type **long**.

Cg 1381.1

If more than one type specifier occurs in a declaration the order used shall be one of those listed in the Standard.

---

type specifiers possible sets of

1382

--~ **void**
--~ **char**
--~ **signed char**
--~ **unsigned char**
--~ **short**, **signed short**, **short int**, or **signed short int**
--~ **unsigned short**, or **unsigned short int**
--~ **int**, **signed**, or **signed int**
--~ **unsigned**, or **unsigned int**
--~ **long**, **signed long**, **long int**, or **signed long int**
--~ **unsigned long**, or **unsigned long int**
--~ **long long**, **signed long long**, **long long int**, or **signed long long int**
--~ **unsigned long long**, or **unsigned long long int**
--~ **float**
--~ **double**

```
---  long double
---  _Bool

---  float _Complex
---  double _Complex
---  long double _Complex
---float _Imaginary
---double _Imaginary
---long double _Imaginary
---  struct or union specifier
---  enum specifier
---  typedef name
```

Commentary

The keyword **signed** can only make a difference to the type in two cases (depending on implementation-defined behavior it may not make any difference), (1) when it appears with **char**, and (2) when it appears in the declaration of a bit-field. However, support for its use creates a symmetry with the keyword **unsigned**.

516 char range, representation and behavior
1387 bit-field int

Rationale

Some pre-C89 implementations allowed type specifiers to be added to a type defined using **typedef**. Thus

```
typedef short int small;
unsigned small x;
```

would give x the type **unsigned short int**. The C89 Committee decided that since this interpretation may be difficult to provide in many implementations, and since it defeats much of the utility of **typedef** as a data abstraction mechanism, such type modifications are invalid.

The wording was changed by the response to DR #207.

C90

Support for the following is new in C99:

— **long long**, **signed long long**, **long long int**, or **signed long long int**
— **unsigned long long**, or **unsigned long long int**
— **_Bool**
— **float _Complex**
— **double _Complex**
— **long double _Complex**

Support for the *no type specifiers* set, in the **int**, **signed**, **signed int** list has been removed in C99.

```
1  extern x;   /* strictly conforming C90  */
2              /* constraint violation C99 */
3  const  y;   /* strictly conforming C90  */
4              /* constraint violation C99 */
5         z;   /* strictly conforming C90  */
6              /* constraint violation C99 */
7         f(); /* strictly conforming C90  */
8              /* constraint violation C99 */
```

C++

The list of combinations, given above as being new in C99 are not supported by C++.

Like C99, the C++ Standard does not require a translator to provide an implicit function declaration returning **int** (footnote 80) being supplied for a missing type specifier.

Other Languages

Most languages use only one sequence of tokens to represent a given type and have a relatively small number of different integer types (often only one). Cobol allows the number of digits used in the representation of decimal types to be specified and many Fortran implementations support more than one integer type. Languages that support integer subranges allow millions of integer types to be defined, although implementations invariably map these to the same underlying representations that are used by C implementations. PL/1 uses a precision specifier rather than keywords to indicate the number of bits (or digits) in various types (e.g., `BINARY(16)` specifies a binary (integer) type represented in nine bits).

Common Implementations

Support for **long long** existed in some vendors implementations for a number of years before C99 was ratified. The set **short double** was supported as a synonym for **float** in a few prestandard implementations. Some implementations continue to support this set for backwards compatibility.

Table 1382.1: Occurrence of *type-specifier* sequences (as a percentage of all type specifier sequences; cut-off below 0.1%). Based on the visible form of the `.c` files.

Type Specifier Sequence	%	Type Specifier Sequence	%
int	39.9	**long**	2.2
void	24.3	**unsigned**	1.6
char	15.6	**unsigned short**	0.9
unsigned long	6.2	**float**	0.6
unsigned int	4.0	**short**	0.5
unsigned char	3.4	**double**	0.5

The type specifiers ~~_Complex and _Imaginary~~ shall not be used if the implementation does not provide ~~those~~ complex types.[102)] 1383

Commentary

This is a constraint that depends on a feature being available in the implementation being used.

The wording was changed by the response to DR #207.

C90

Support for this type specifier is new in C99.

C++

Support for these type specifiers is new in C99 and are not specified as such in the C++ Standard. The header **<complex>** defines template classes and associated operations whose behavior provides the same functionality as that provided, in C, for objects declared to have type **_Complex**. There are no equivalent definitions for **_Imaginary**.

Other Languages

implementation two forms complex types 92 500

Cobol and Fortran (77) support optional type specifiers. Only a few languages support complex types.

Coding Guidelines

Source code that uses the specifier **_Complex** or **_Imaginary** has a dependence on implementations providing the necessary support. Translating source code containing one of these type specifiers, using an implementation that does not support them, will result in a diagnostic being issued. The behavior is either defined, or a diagnostic is issued. There is no benefit from having a guideline recommendation.

Semantics

1384 Specifiers for structures, unions, and enumerations are discussed in 6.7.2.1 through 6.7.2.3.

Commentary

See subclauses 6.7.2.1 through 6.7.2.3 for the discussion.

1390 struct/union syntax
1439 enumeration specifier syntax
1454 type contents defined once

1385 Declarations of typedef names are discussed in 6.7.7.

Commentary

See subclause 6.7.7 for the discussion

1629 typedef name syntax

1386 The characteristics of the other types are discussed in 6.2.5.

Commentary

See subclause 6.2.5 for the discussion

472 types

1387 Each of the comma-separated sets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **int** designates the same type as **signed int** or the same type as **unsigned int**.

bit-field int

Commentary

That is, the sets appearing on the same line designate the same type.

The treatment of bit-field types is not quiet the same as the treatment of character types, although it has been influenced by similar historical factors (variations between early implementations). While an **int** bit-field is the same type as either the signed or unsigned forms, the type **char** is a different type from the signed and unsigned character types (although it is capable of representing the same range of values as one of them).

515 character types
516 char range, representation and behavior

C90

Each of the above comma-separated sets designates the same type, except that for bit-fields, the type ***signed int*** *(or* ***signed****) may differ from* ***int*** *(or no type specifiers).*

C++

Rather than giving a set of possibilities, the C++ Standard lists each combination of specifiers and its associated type (Table 7).

Other Languages

Languages that offer some mechanism for controlling the number of storage bits allocated to an object, do not usually allow that mechanism to change the underlying type of the object.

Common Implementations

Many translators support an option that enables the developer to control how this choice is made.

Coding Guidelines

Most operands having a bit-field type promote to the type **int**. The only bit-field type that does not is one declared with the type specifier **unsigned int**, using a width that is the same as that of the type **unsigned int** (which can occur in source designed to be ported to a variety of architectures through the use of macros to defined the field width). While occurrences of operands having a bit-field type may promote to **int**, the underlying value representation supports a limited range of values. It is possible that translating the same source using different implementations will result in a dramatic change in the range of representable values (in the case of signed to unsigned, negative values being unrepresentable).

Cg 1387.1 The declaration of a bit-field type shall always include one of the type specifiers **signed** or **unsigned**.

footnote 102

102) ~~101)Implementations are not required to provide imaginary types.~~ Freestanding implementations are not required to provide complex types. 1388

Commentary

conforming 94 freestanding implementation

Complex types are one of many features a freestanding implementation does not have to provide. Not being able to accept a program that uses such types does not affect such an implementation's conformance status.

The wording was changed by the response to DR #207.

C90

Support for complex types is new in C99.

C++

There is no specification for imaginary types (in the **<complex>** header or otherwise) in the C++ Standard.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.7). 1389

6.7.2.1 Structure and union specifiers

struct/union syntax

1390

struct-or-union-specifier:
struct-or-union *identifier*$_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union *identifier*
struct-or-union:
struct
union
struct-declaration-list:
struct-declaration
struct-declaration-list *struct-declaration*
struct-declaration:
specifier-qualifier-list *struct-declarator-list* **;**
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{opt}$
type-qualifier *specifier-qualifier-list*$_{opt}$
struct-declarator-list:
struct-declarator
struct-declarator-list **,** *struct-declarator*
struct-declarator:
declarator
declarator$_{opt}$ **:** *constant-expression*

Commentary

The use of **{** and **}** to delimit the sequence of members is consistent with them being used to delimit the sequence of statements in a compound statement.

C++

The C++ Standard uses the general term *class* to refer to these constructs. This usage is also reflected in naming of the nonterminals in the C++ syntax. The production *struct-or-union* is known as *class-key* in C++ and also includes the keyword **class**. The form that omits the brace enclosed list of members is known as an *elaborated-type-specifier* (7.1.5.3) in C++.

Other Languages

Some languages (e.g., Pascal) use the keyword **record** to denote both types and the keyword **end** to denote the end of the declaration (which is consistent with the delimiters used in these languages for compound statements, e.g., **begin/end**). The following example shows how Ada permits developers to specify which word, and bits within a word, are occupied by a member.

```
WORD : constant := 4;

type STATUS_WORD is
   record
   system_mask : array(0..7) of BOOLEAN;
   key         : INTEGER range 0..3;
   inst_addr   : ADDRESS;
   end record;

for STATUS_WORD use
   record at mod 8  -- address at which object is allocated storage
   system_mask at 0*WORD range  0.. 7; -- occupies bits 0 through 7, inclusive
   key         at 0*WORD range 10..11; -- occupies 2 bits
   inst_addr   at 1*WORD range  8..31; -- allocate member in second word
   end record;
```

The following is an example of a record declaration in Cobol.

```
        01 WEEKLY-SALES
           05 SALES-TABLE          OCCURS 50 TIMES.
              10 PRODUCT-CODE         PIC X(10).
              10 PRODUCT-PRICE        PIC S9(4)V99.
              10 PRODUCT-COUNT        PIC XXX.
                 88 LAST_PRODUCT                    VALUE "999".
```

Common Implementations

Some translators (`gcc`, and Plan 9 C[1350]) support the concept of anonymous structure and union members. In the example below the member name of a union type is omitted. When resolving names along a chain of selections a translator has to deduce when a member of a union is intended.

```
struct {
       int mem1;
       union {
              char mem2;
              long mem3;
             }; /* anonymous */
       } x;

void f(void)
{
x.mem3=3;
}
```

Cyclone C[668] supports tagged union types, using the keyword **tunion**, which contain information that identifies the current member.

Coding Guidelines

The visibility issues associated with object identifiers declared in an *init-declarator* also apply to identifiers declared in a *struct-declarator*.

1348.1 init-declarator one per source line

Cg 1390.1 No more than one *struct-declarator* shall occur on each visible source code line.

The discussion on the layout of declarators also applies to declarators in structure definitions. Deciding which members belong in which structure type can involve trade-offs amongst many factors. This issue is discussed in more detail elsewhere.

1358 declarator list of
299 limit struct/union nesting

Usage

A study by Sweeney and Tip[1324] of C++ applications found that on average 11.6% of members were dead (i.e., were not read from) and that 4.4% of object storage space was occupied by these dead data members. Usage information on member names and their types is given elsewhere (see Table 443.1 and Table 443.2).

Table 1390.1: Number of occurrences of the given token sequence. Based on the visible source of the `.c` files (`.h` files in parentheses).

Token Sequence	Occurrences	Token Sequence	Occurrences
enum {	456 (1,591)	**struct** id **;**	76 (13,384)
enum id **;**	0 (0)	**struct** id id	122,974 (27,589)
enum id **{**	474 (1,059)	**union {**	297 (725)
enum id id	2,922 (633)	**union** id **;**	0 (11)
struct {	1,567 (6,503)	**union** id **{**	105 (2,624)
struct id **{**	4,407 (1,311)	**union** id id	330 (231)

Constraints

member not types

A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; 1391

Commentary

Allowing structure or union members to have an incomplete type creates complications (the types would have to be completed at some point and a mechanism would need to be created to perform it) for little benefit. Requiring that the members have a complete type also simplifies the job of the translator. It becomes be possible to assign offsets, relative to the first member, to each member as it is encountered. The exception is for the last member where there is no following member requiring an offset. The requirement on there being at least one other named member, when the last member has an incomplete type, arises because of how the size of the declared type is calculated.

flexible array member 1430

structure size with flexible member 1432

The intent of the exception case is to support an implementation model where the storage for the last member follows the storage allocated to the other members in the same type (i.e., the storage for the last member does not need to be allocated in a separate storage area, with references to the member being implicitly dereferenced), but the amount allocated is decided during program execution.

Members having function type are supported in object-oriented languages. Although a proposal was submitted to the C committee, WG14/{N424, N445, N446, N447}, to add classes to C99 the additional complexity was not considered to be in the spirit of C.

spirit of C 14

C90

Support for the exception on the last named member is new in C99.

C++

It is a design feature of C++ that class types can contain incomplete and function types. Source containing instances of such constructs is making use of significant features of C++ and there is unlikely to be any expectation of being able to successfully process it using a C translator.

The exception on the last named member is new in C99 and this usage is not supported in the C++ Standard.

The following describes a restriction in C++ that does not apply in C.

annex C.1.7p3

Change: *In C++, a* **`typedef`** *name may not be redefined in a class declaration after being used in the declaration*

Example:

```
typedef int I;
struct S {
          I i;
          int I;  // valid C, invalid C++
         };
```

Rationale: *When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.*

Other Languages

Other languages follow C in requiring members to be declared with complete types. Object-oriented language allow class definitions to include function definitions.

Example

An awkward edge case that needs to be handled by any implementation claiming to support incomplete member types.

```
1  struct x { struct y y; }; /* Constraint violation in Standard C. */
2  struct y { struct x x; };
```

1392 such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

Commentary

This requirement applies to the exception case and it describes contexts that all require types of known size. Requiring support for this usage would require the array storage for the last member to be allocated somewhere other than after the storage for the member that preceded it.

1393 The expression that specifies the width of a bit-field shall be an integer constant expression that has a nonnegative value that shall not exceed the ~~number~~width of ~~bits in~~ an object of the type that ~~is~~would be specified ~~if~~were the colon and expression ~~are~~ omitted.

bit-field maximum width

Commentary

The intent of a bit-field declaration is to specify the number of bits in a types value representation, a negative value has no meaning. Objects having a bit-field type can be used wherever expressions having certain integer types can be used. This usage can only be guaranteed to be defined if bit-fields do not have a width greater than these types. Implementations are required to supported a width of at least 1 for bit-fields of type **_Bool**. However, they may also support greater widths, up to a maximum of CHAR_BIT.

575 bit-field value is m bits

670 expression wherever an int may be used

666 _Bool rank

The wording was changed by the response to DR #262 (making it clear that any padding bits are not counted).

C90

The C90 wording ended with " . . . of bits in an ordinary object of compatible type.", which begs the question of whether bit-fields are variants of integer types or are separate types.

C++

The C++ issues are discussed elsewhere.

575 bit-field value is m bits

Coding Guidelines

Specifying a width equal to the number of bits that would have been used in the value representation, had the colon and expression been omitted, would appear to be redundant. However, the source may be translated

using a variety of different implementations in different host environments and the equality in the number of bit-field int 1387 bits used may not apply to all of them. Also, when declared using plain **int** the signedness of a bit-fields type will be the same as other bit-fields declared using plain **int**. How the width is specified (through the replacement of a macro name, or a constant expression) ties in with the general software engineering topic of source configuration and is outside the scope of these coding guidelines.

If the value is zero, the declaration shall have no declarator. 1394

Commentary

bit-field zero width 1415 This construct is discussed elsewhere.

C++

9.6p2 *Only when declaring an unnamed bit-field may the* `constant-expression` *be a value equal to zero.*

Source developed using a C++ translator may contain a declaration of a zero width bit-field that include a declarator, which will generate a constraint violation if processed by a C translator.

```
struct {
        int mem_1;
        unsigned int mem_2:0; //                  no diagnostic required
                              /* constraint violation, diagnostic required */
       } obj;
```

There is an open C++ DR (#057) concerning the lack of a prohibition against declarations of the form:

```
union {int : 0;} x;
```

---

bit-field shall have type

A bit-field shall have a type that is a qualified or unqualified version of **_Bool**, **signed int**, **unsigned int**, or some other implementation-defined type. 1395

Commentary

In this context use of the type specifier **int** is equivalent to either **signed int** or **unsigned int**. It is not a different type (as **char** is from **signed char** and **unsigned char**) and so need not appear in the list here.

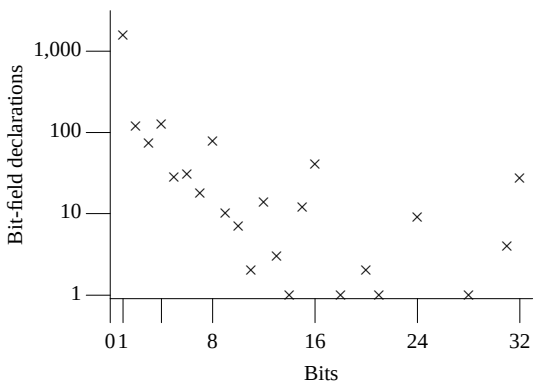


Figure 1393.1: Number of bit-field declarations specifying a given number of bits. Based on the translated form of this book's benchmark programs. (Declarations encountered in any source or header file were only counted once, the contents of system headers were ignored.)

The phrase *implementation-defined type* refers to any other standard integer type that an implementation chooses to support in a bit-field declaration, or any an implementation-defined extended integer type. Using " . . . some other implementation-defined type." is making use of implementation-defined behavior.

493 standard integer types
493 extended integer types

C90

The following wording appeared in a semantics clause in C90, not a constraint clause.

> *A bit-field shall have a type that is a qualified or unqualified version of one of* **int**, **unsigned int**, *or* **signed int**.

Programs that used other types in the declaration of a bit-field exhibited undefined behavior in C90. Such programs exhibit implementation-defined behavior in C99.

C++

> *A bit-field shall have integral or enumeration type (3.9.1).* 9.6p3

Source developed using a C++ translator may contain bit-fields declared using types that are a constraint violation if processed by a C translator.

```
enum E_TAG {a, b};

struct {
        char m_1  : 3;
        short m_2 : 5;
        long m_3  : 7;
        enum E_TAG m_4 : 9;
       } glob;
```

Other Languages

Languages that provide a mechanism for specifying the layout of aggregate types (e.g., Ada and CHILL) do not usually place restrictions on the types that the objects may have. Although they may limit the range of possible widths for some types.

Common Implementations

Some implementations support bit-fields declared using other integer types (`gcc` and the SVR4 C compiler). This usage is not purely to intended to support value representations containing a greater number of bits. Implementations often use the alignment of the type specifier used as the alignment of the addressable storage unit to be used to hold the bit-field. For instance, if the alignment used for the type **short** was 2 and that for **int** 4, then the alignment of the addressable storage unit used to hold bit-fields declared using the two types would be 2 and 4 respectively.

While some processors (e.g., Intel Pentium) include instructions for operating on what are sometimes called *packed floating-point* values, the representation used for these *packed* values is the same as that used for their *unpacked* form. The term *packed* being applied in the case where instructions operate on two or more floating-point values as a single unit (e.g., four 32-bit values, represented in 128 bits, are added to another four 32-bit values being represented in 128 bits).

Coding Guidelines

The discussion leading to the guideline recommendation dealing with the use of a single integer type is applicable here.

480.1 object int type only

Use of implementation-defined integer types, whose rank is less than that of **int**, almost certainly implies that a developer is trying to control the layout of objects in storage. While there is a guideline recommendation

1354 storage layout

covering this usage, bit-fields are often treated as a special case. There are sometimes application domain storage layout requirements. For instance, interpreting different sequences of bits at various bit offsets, from some starting point, as corresponding to different value representations.

extensions 95.1 cost/benefit

An alternative to declaring members as bit-field types is to declare them as non-bit-field integer types and use bitwise operations to extract the required sequence of value bits. This approach removes any dependency on how implementations allocate bit-field members to storage locations. The costs are a potential increase in effort needed to comprehend the source (unless the details of the bit sequence selection are hidden behind a macro call) and a potential increase in execution time (it is likely to be easier to optimize the generated machine code for an access using a member selection operator than a sequence of developer selected bitwise operations). Provided macros are used to hide the implementation details, neither of these costs is likely to be significant.

A developer may want to minimize the amount of storage used by a type and have no interest in the actual storage layout selected. Recommending against the use of other implementation-defined type because storage minimization is not guaranteed on other implementations is a rather rigid interpretation of guidelines.

storage 1354 layout

Dev 95.1 A bit-field may be declared using a type whose rank is less than that of **int** provided no use is made of member layout information.

Dev 95.1 A bit-field may be declared using a type whose rank is greater than that of **int**.

Semantics

As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap. 1396

Commentary

These issues are discussed elsewhere.

structure type 530 sequentially allocated objects
union type 531 overlapping members

C++

This requirement can be deduced from 9.2p12 and 9.5p1.

Structure and union specifiers have the same form. 1397

Commentary

They have the same syntactic form (the only syntactic difference is the keyword used). As types both structures and unions are either identified by their tag name, a typedef name, or are anonymous, within individual translation units (their members are not used to determine type compatibility). However, between different translation units, type compatibility includes requirements on the members in a structure or union type.

types 472
footnote 650 46
compatible 633 separate translation units

C++

The C++ Standard does not make this observation.

Other Languages

In some languages (e.g., Pascal and Ada) a variant record (union) can only occur within a record (structure) declaration. It is not possible to declare the equivalent of a union at the outer most declaration level.

The keywords **struct** and **union** indicate that the type being specified is, respectively, a structure type or a union type. 1398

Commentary

This sentence was added by the response to DR #251. The reason for it is discussed elsewhere.

tag name 1455 same struct, union or enum

1399 The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit.

Commentary

This wording specifies that the form: *struct-or-union identifier*$_{opt}$ **{** *struct-declaration-list* **}** declares a new type. Other forms of structure declaration that omit the braces either declare an identifier as a tag or refer to a previous declaration.

Other Languages

Whether or not a structure or union type definition is a new type may depend on a languages type compatibility rules. Languages that use structural equivalence may treat different definitions as being the same type (usually employing rules similar to those used by C for type compatibility across translation units).

633 compatible separate translation units

1400 The struct-declaration-list is a sequence of declarations for the members of the structure or union.

Commentary

Say in words what is specified in the syntax.

1401 If the struct-declaration-list contains no named members, the behavior is undefined.

Commentary

The syntax does not permit the *struct-declaration-list* to be empty. However, it is possible for members to be unnamed bit-fields.

1414 bit-field unnamed

C++

An object of a class consists of a (possibly empty) sequence of members and base class objects. 9p1

Source developed using a C++ translator may contain class types having no members. This usage will result in undefined behavior when processed by a C translator.

Other Languages

The syntax of languages invariably requires at least one member to be declared and do not permit zero sized types to be defined.

Common Implementations

Most implementations issue a diagnostic when they encounter a *struct-declaration-list* that does not contain any named members. However, many implementations also implicitly assume that all declared objects have a nonzero size and after issuing the diagnostic may behave unpredictably when this assumption is not met.

Coding Guidelines

A discussion of the cost/benefit of the use of such a construct (perhaps based on its apparent harmlessness versus possible unpredictable translator behavior). In practice occurrences of this construct are rare (until version 3.3.1 `gcc` reported "internal compiler error" for many uses of objects declared with such types) and is not worth a guideline recommendation.

Example

```
1  #include <stdio.h>
2
3  struct S {
4            int : 0;
```

```
5           };
6
7   void f(void)
8   {
9   struct S arr[10];
10
11  printf("arr contains %d elements\n", sizeof(arr)/sizeof(struct S));
12  }
```

struct type incomplete until

The type is incomplete until after the **}** that terminates the list. 1402

Commentary

tag 1458 incomplete until

This sentence is a special case of one discussed elsewhere.

Example

```
1   struct S {
2             int m1;
3             struct S m2; /* m2 refers to an incomplete type (a constraint violation). */
4           }             /* S is complete now. */;
5   struct T {
6             int m1;
7           } x = { sizeof(struct T) }; /* sizeof a completed type. */
```

In the second definition the closing **}** (the one before the x) completes the type and the **sizeof** operator can be applied to the type.

struct member type

A member of a structure or union may have any object type other than a variably modified type.[103] 1403

Commentary

member 1391 not types

Other types are covered by a constraint. As the discussion for that C sentence points out, the intent is to enable a translator to assign storage offsets to members at translation time. Apart from the special case of the last member, the use of variably modified types would prevent a translator assigning offsets to members (because their size is not known at translation time).

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard.

Other Languages

Java uses references for all non-primitive types. Storage for members having such types need not be allocated in the class type that contains the member declaration and there is no requirement that the number of elements allocated to a member having array type be known at translation time.

Table 1403.1: Occurrence of structure member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
int	15.8	**unsigned short**	7.7	**char** *	2.3	**void** *()	1.3
other-types	12.7	**struct**	7.2	**enum**	1.9	**float**	1.2
unsigned char	11.1	**unsigned long**	5.2	**long**	1.8	**short**	1.0
unsigned int	10.4	**unsigned**	4.0	**char**	1.8	**int** *()	1.0
struct *	8.8	**unsigned char** []	3.1	**char** []	1.5		

Table 1403.2: Occurrence of union member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
`struct`	46.9	`unsigned int`	3.8	`double`	1.9	`char []`	1.3
other-types	11.3	`char *`	2.8	`enum`	1.7	`union *`	1.1
`struct *`	8.3	`unsigned long`	2.4	`unsigned char`	1.5		
`int`	6.0	`unsigned short`	2.1	`struct []`	1.3		
`unsigned char []`	4.3	`long`	2.1	`( struct * ) []`	1.3		

1404 In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any).

Commentary

The ability to declare an object that consists of a specified number of bits is only possible inside a structure or union type declaration.

Other Languages

Some languages (e.g., CHILL) provide a mechanism for specifying how the elements of arrays are laid out and the number of bits they occupy. Languages in the Pascal family support the concept of subranges. A subrange allows the developer to specify the minimum and maximum range of values that an object needs to be able to represent. The implementation is at liberty to allocate whatever resources are needed to satisfy this requirement (some implementations simply allocate an integers worth of storage, while others allocate the minimum number of bytes needed).

Coding Guidelines

Why would a developer want to specify the number of bits to be used in an object representation? This level of detail is usually considered to be a low level implementation information. The following are possible reasons for this usage include:

- *Minimizing the amount of storage used by structure objects.* This remains, and is likely to continue to remain, an important concern in applications where available storage is very limited (usually for cost reasons).
- *There is existing code*, originally designed to run in a limited storage environment. The fact that storage requirements are no longer an issue is rarely a cost effective rationale for spending resources on removing bit-field specifications from declarations.
- *Mapping to a hardware device.* There are often interfaced via particular storage locations (organized as sequences of bits), or transfer data is some packed format. Being able to mirror the bit sequences of the hardware using some structure type can be a useful abstraction (which can require the specification of the number of bits to be allocated to each object).
- *Mapping to some protocol imposed layout of bits.* For instance, the fields in a network data structure (e.g., TCP headers).

The following are some of the arguments that can be made for not using bit-fields types:

- Many of the potential problems associated with objects declared to have an integer type, whose rank is less than `int`, also apply to bit-fields. However, one difference between them is that developers do not habitually use bit-fields, to the extent that character types are used. If developers don't use bit-fields out of habit, but put some thought into deciding that their use is necessary a guideline recommendation would be redundant (treating guideline recommendations as prepacked decision aids).
- It is making use of representation information.

480.1 object int type only
0 coding guidelines introduction
569 types representation

- The specification of bit-field types involves a relatively large number of implementation-defined behaviors, dealing with how bit-fields are allocated in storage. However, recommending against the use of bit-fields only prevents developers from using one of the available techniques for accessing sequences of bits within objects. It is not obvious that bit-fields offer the least cost/benefit of all the available techniques (although some coding guideline documents do recommend against the use of bit-fields).

Dev 569.1 Bit-fields may be used to interface to some externally imposed storage layout requirements.

---

bit-field

Such a member is called a *bit-field*;[104)] 1405

Commentary

This defines the term *bit-field*. Common usage is for this term to denote bit-fields that are named. The less frequently used unnamed bit-fields being known as *unnamed bit-fields*.

bit-field unnamed 1414

Other Languages

Languages supporting such a type use a variety of different terms to describe such a member.

---

its width is preceded by a colon. 1406

Commentary

Specifying in words the interpretation to be given to the syntax.

Other Languages

Declarations in languages in the Pascal family require the range of values, that need to be representable, to be specified in the declaration. The number of bits used is implementation-defined.

---

bit-field interpreted as

A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.[105)] 1407

Commentary

Both the value and object representation use the same number of bits. In some cases there may be padding between bit-fields, but such padding cannot be said to belong to any particular member.

value representation 595

object representation 574

C++

The C++ Standard does not specify (9.6p1) that the specified number of bits is used for the value representation.

Coding Guidelines

Using a symbolic name to specify the width might reduce the effort needed to comprehend the source and reduce the cost making changes to the value in the future.

symbolic name 822

If the value 0 or 1 is stored into a nonzero-width bit-field of type **_Bool**, the value of the bit-field shall compare equal to the value stored. 1408

Commentary

This is a requirement on the implementation. It is implied by the type **_Bool** being an unsigned integer type (for signed types a single bit bit-field can only hold the values 0 and -1). These are also the only two values that are guaranteed to be represented by the type **_Bool**.

standard unsigned integer 487

_Bool large enough to store 0 and 1 476

C90

Support for the type **_Bool** is new in C99.

---

bit-field addressable storage unit

An implementation may allocate any addressable storage unit large enough to hold a bit-field. 1409

Commentary

There is no requirement on implementations to allocate the smallest possible storage unit. They may even allocate more bytes than `sizeof(int)`.

Other Languages

Languages that support some form of object layout specification often require developers to specify the storage unit and the bit offset, within that unit, where the storage for an object starts.

1390 struct/union syntax

Common Implementations

Many implementations allocate the same storage unit for bit-fields as they do for the type **`int`**. The only difference being that they will often allocate storage for more than one bit-field in such storage units. Implementations that support bit-field types having a rank different from **`int`** usually base the properties of the storage unit used (e.g., alignment and size) on those of the type specifier used.

1410 bit-field packed into
1395 bit-field shall have type

Coding Guidelines

Like other integer types, the storage unit used to hold bit-field types is decided by the implementation. The applicable guidelines are the same.

1395 bit-field shall have type
569.1 representation information using

Example

```
#include <stdio.h>

struct {
        char m_1;
        signed int m_2 :3;
        char m_3;
       } x;

void f(void)
{
if ((&x.m_3 - &x.m_1) == sizeof(int))
   printf("bit-fields probably use the same storage unit as int\n");
if ((&x.m_3 - &x.m_1) == 2*sizeof(int))
   printf("bit-fields probably use the same storage unit and alignment as int\n");
}
```

1410 If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit.

bit-field packed into

Commentary

This is a requirement on the implementation. However, any program written to verify what the implementation has done, has to make use of other implementation-defined behavior. This requirement does not guarantee that all adjacent bit-fields will be packed in any way. An implementation could choose its addressable storage unit to be a byte, limiting the number of bit-fields that it is required to pack. However, if the storage unit used by an implementation is a byte, this requirement means that all members in the following declaration must allocated storage in the same byte.

```
struct {
        int mem_1 : 5;
        int mem_2 : 1;
        int mem_3 : 2;
       } x;
```

C++

This requirement is not specified in the C++ Standard.

9.6p1 *Allocation of bit-fields within a class object is implementation-defined.*

---

bit-field overlaps storage unit

If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. 1411

Commentary

spirit of C 14

One of the principles that the C committee derived from the spirit of C was that an operation should not expand to a surprisingly large amount of machine code. Reading a bit-field value is potentially three operations; load value, shift right, and zero any unnecessary significant bits. If implementations were required to allocate bit-fields across overlapping storage units, then accessing such bit-fields is likely to require at least twice as many instructions on processors having alignment restrictions. In this case it would be necessary to load values from the two storage units into two registers, followed by a sequence of shift, bitwise-AND, and bitwise-OR operations. This wording allows implementation vendors to chose whether they want to support this usage, or leave bits in the storage unit unused.

alignment 39

Other Languages

Even languages that contain explicit mechanisms for specifying storage layout sometimes allow implementations to place restrictions on how objects straddle storage unit boundaries.

Common Implementations

Implementations that do not have alignment restrictions can access the appropriate bytes in a single load or store instruction and do not usually include a special case to handle overlapping storage units. Some processors include instructions[968] that can load/store a particular sequence of bits from/to storage.

representation information using 569.1

Coding Guidelines

The guideline recommendation dealing with the use of representation information are applicable here.

Example

The extent to which any of the following members are put in the same storage unit is implementation-defined.

```
struct T {
           signed int m_1 :5;
           signed int m_2 :5; /* Straddles an 8-bit boundary. */
           signed int m_3 :5;
           signed int m_4 :5; /* Straddles a 16-bit boundary. */
           signed int m_5 :5;
           signed int m_6 :5;
           signed int m_7 :5; /* Straddles a 32-bit boundary. */
         };
```

The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. 1412

Commentary

byte 53 addressable unit
object 570 contiguous sequence of bytes
member 1422 address increasing
unary & 1088 operand constraints

An implementation is required to chose one of these two orderings The standard does not define an order for bits within a byte, or for bytes within multibyte objects. Either of these orderings is consistent with the relative order of members required by the Standard.

It is not possible to take the address of an object having a bit-field type, and so bit-field member ordering cannot be deduced using pointer comparisons. However, the ordering can be deduced using a union type.

Common Implementations

While there is no requirement that the ordering be the same for each sequence of bit-field declarations (within a structure type), it would be surprising if an implementation used a different ordering for different declarations. Many implementations use the allocation order implied by the order in which bytes are allocated within multibyte objects.

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

569.1 representation information using

Example

```
/*
 * The member bf.m_1 might overlap the same storage as m_4[0] or m_4[1]
 * (using a 16-bit storage unit).  It might also be the most significant
 * or least significant byte of m_3 (using int as the storage unit).
 */
union {
       struct {
                signed int m_1 :8;
                signed int m_2 :8;
               } bf;
       int m_3;
       char m_4[2];
      } x;
```

1413 The alignment of the addressable storage unit is unspecified.

alignment addressable storage unit

Commentary

This behavior differs from that of the non-bit-field members, which is implementation-defined.

1421 member alignment

C++

The wording in the C++ Standard refers to the bit-field, not the addressable allocation unit in which it resides. Does this wording refer to the alignment within the addressable allocation unit?

Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. 9.6p1

Common Implementations

Implementations that support bit-field types having a rank different from **int** usually base the properties of the alignment used on those of the type specifier used.

1395 bit-field shall have type

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

569.1 representation information using

1414 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[106)]

bit-field unnamed

Commentary

Memory mapped devices and packed data sometimes contains sequences of bits that have no meaning assigned to them (sometimes called *holes*). When creating a sequence of bit-fields that map onto the meaningful values any holes also need to be taken into account. Unnamed bit-fields remove the need to create an *anonymous* name (sometimes called a *dummy* name) to denote the bit sequences occupied by the holes. In some cases the design of a data structure might involve having some spare bits, between certain members, for future expansion.

Other Languages

Languages that support some form of layout specification usually use a more direct method of specifying where to place objects (using bit offset and width). It is not usually necessary to specify where the holes go.

Coding Guidelines

Any value denoted by the sequence of bits specified by an unnamed bit-field is not accessible to a conforming program. The usage is purely associated with specifying representation details. There is no minimization of storage usage justification and the guideline recommendation dealing with the use of representation information is applicable here.

representation information using 569.1

---

bit-field zero width

As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed. 1415

Commentary

This special case provides an additional, developer accessible, mechanism for controlling the layout of bit-fields in structure types (it has no meaningful semantics for members of union types). It might be thought that this special case is redundant, a developer either working out exactly what layout to use for a particular implementation or having no real control over what layout gets used in general. However, if an implementation supports the allocation of bit-fields across adjacent units a developer may be willing to trade less efficient use of storage for more efficient access to a bit-field. Use of a zero width bit-field allows this choice to be made.

bit-field overlaps storage unit 1411

---

footnote 103

103) A structure or union can not contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3. 1416

Commentary

It would have been possible for the C committee to specify that members could have a variably modified type. The reasons for not requiring such functionality are discussed elsewhere.

variable modified only scope 1569

C90

Support for variably modified types is new in C99.

C++

Variably modified types are new in C99 and are not available in C++.

---

footnote 104

104) The unary & (address-of) operator cannot be applied to a bit-field object; 1417

Commentary

Such an occurrence would be a constraint violation.

unary & operand constraints 1088

---

thus, there are no pointers to or arrays of bit-field objects. 1418

Commentary

The syntax permits the declaration of such bit-fields and they are permitted as implementation-defined extensions. The syntax for declarations implies that the declaration:

bit-field shall have type 1395

```
1  struct {
2          signed int abits[32] : 1;
3          signed int *pbits : 3;
4          } vector;
```

declares `abits` to have type array of bit-field, rather than being a bit-field of an array type (which would also violate a constraint). Similarly `pbits` has type pointer to bit-field.

bit-field shall have type 1395

One of the principles that the C committee derived from the spirit of C was that an operation should not

spirit of C 14

expand to a surprisingly large amount of machine code. Arrays of bit-fields potentially require the generation of machine code to perform relatively complex calculations, compared to non-bit-field element accesses, to calculate out the offset of an element from the array index, and to extract the necessary bits.

The C pointer model is based on the byte as the smallest addressable storage unit. As such it is not possible to express the address of individual bits within a byte.

53 byte addressable unit

Other Languages

Some languages (e.g., Ada, CHILL, and Pascal) support arrays of objects that only occupy some of the bits of a storage unit. When translating such languages, calling a library routine that extracts the bits corresponding to the appropriate element is often a cost effective implementation technique. Not only does the offset need to be calculated from the index, but the relative position of the bit sequence within a storage unit will depend on the value of the index (unless its width is an exact division of the width of the storage unit). Pointers to objects that do not occupy a complete storage unit are rarely supported in any language.

1419 105) As specified in 6.7.2 above, if the actual type specifier used is **int** or a typedef-name defined as **int**, then it is implementation-defined whether the bit-field is signed or unsigned.

footnote 105

Commentary

This issue is discussed elsewhere.

1387 bit-field int

C90

This footnote is new in C99.

1420 106) An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

footnote 106

Commentary

Bit-fields, named or otherwise, in general are useful for padding to conform to externally imposed layouts (however, giving names to the padding bits can provide useful source code information).

1421 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

member alignment

Commentary

The standard does not require the alignment of other kinds of objects to be documented. Developers sometimes need to be able to calculate the offsets of members of structure types (the `offsetof` macro was introduced into C90 to provide a portable method of obtaining this information). Knowing the size of each member, the relative order of members, and their alignment requirements is invariably sufficient information (because implementations insert the minimum padding between members necessary to produce the required alignment).

1422 member address increasing

While all members of the same union object have the same address, the alignment requirements on that address may depend on the types of the members (because of the requirement that a pointer to an object behave the same as a pointer to the first element of an array having the same object type).

1207 pointer to union members compare equal
1165 additive operators pointer to object

C++

The C++ Standard specifies (3.9p5) that the alignment of all object types is implementation-defined.

Other Languages

Most languages do not call out a special case for the alignment of members.

Common Implementations

Most implementations use the same alignment requirements for members as they do for objects having automatic storage duration. It is possible for the offset of a member having an array type to depend on the number of elements it contains. For instance, the Motorola 56000 supports pointer operations on circular buffers, but requires that the alignment of the buffer be a power of 2 greater than or equal to the buffer size.

39 alignment
39 Motorola 56000

Coding Guidelines

storage 1354 layout

The discussion on making use of storage layout information is applicable here.

---

member address increasing

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. 1422

Commentary

structure 1206 members later compare later

Although not worded as such, this is effectively a requirement on the implementation. It is consistent with a requirement on the result of comparisons of pointers to members of the same structure object. Prior to the publication of the C Standard there were several existing practices that depended on making use of information on the relative order of members in storage; including:

- Accessing individual members of structure objects via pointers whose value had been calculated by performing arithmetic on the address of other members (the `offsetof` macro was invented by the committee to address this need).
- Making use of information on the layout of members to overlay the storage they occupy with other objects.

member 1421 alignment

By specifying this ordering requirement the committee prevented implementations from using a different ordering (for optimization reasons), increasing the chances that existing practices would continue to work as expected (these practices also rely on other implementation-defined behaviors). The cost of breaking existing code and reducing the possibility of being able to predict member storage layout was considered to outweigh any performance advantages that might be obtained from allowing implementations to choose the relative order of members.

C++

The C++ Standard does not say anything explicit about bit-fields (9.2p12).

Other Languages

Few other languages guarantee the ordering of structure members. In practice, most implementations for most languages order members in storage in the same sequences as they were declared in the source code. The **packed** keyword in Pascal is a hint to the compiler that the storage used by a particular record is to be minimized. A few Pascal (and Ada) implementations reorder members to reduce the storage they use, or to change alignments to either reduce the total storage requirements or to reduce access costs for some frequently used members.

Common Implementations

The quantity and quality of analysis needed to deduce when it is possible to reorder members of structures has deterred implementors from attempting to make savings, for the general case, in this area. Some impressive savings have been made by optimizers[741] for languages that do not make this pointer to member guarantee.

Palem and Rabbah[1043] looked at the special case of dynamically allocated objects used to create tree structures. Building a tree structure usually requires the creation of many objects having the same type. A common characteristic of some operations on tree structures is that an access to an object, using a particular member name, is likely to be closely followed by another access to an object using the same member name. Rather than simply reordering members, they separated out each member into its own array, based on dynamic profiles of member accesses (the Trimaran[46] and `gcc` compilers were modified to handle this translation internally; it was invisible to the developer). For instance:

```
1  struct T {
2             int m_1;
3             struct T *next;
4            };
5  /*
```

```
6    * Internally treated as if written
7    */
8   int m_1[4];
9   struct T *(next[4]);
```

dynamically allocating storage for an object having type `struct T` resulted in storage for the two arrays being allocated. A second dynamic allocation request requires no storage to be allocated, the second array element from the first allocation could be used. If tree structures are subsequently walked in an order that is close to the order in which they are built, there is an increased probability that members having the same name will be in the same cache line. Using a modified `gcc` to process seven data intensive benchmarks resulted in an average performance improvement of 24% on Intel Pentium II and III, and 9% on Sun Ultra-Sparc-II.

Franz and Kistler[444] describe an optimization that splits objects across non-contiguous storage areas to improve cache performance. However, their algorithm only applies to strongly typed languages where developers cannot make assumptions about member layout, such as Java.

pointer compressing members

Zhang and Gupta[1511] developed what they called the *common-prefix* and *narrow-data* transformations. These compress 32-bit integer values and 32-bit address pointers into 15 bits. This transformation is dynamically applied (the runtime system checks to see if the transformation can be performed) to the members of dynamically allocated structure objects, enabling two adjacent members to be packed into a 32-bit word (a bit is used to indicate a compressed member). The storage optimization comes from the commonly seem behavior: (1) integer values tend to be small (the runtime system checks whether the top 18 bits are all 1's or all 0's), and (2) that the addresses of the links, in a linked data structure, are often close to the address of the object they refer to (the runtime system checks whether the two address have the same top 17 bits). Extra machine code has to be generated to compress and uncompress members, which increases code size (average of 21% on the user code, excluding linked libraries) and lowers runtime performance (average 30%). A reduction in heap usage of approximate 25% was achieved (the Olden benchmarks were used).

0 Olden benchmark

Coding Guidelines

The order of storage layout of the members in a structure type is representation information that is effectively guaranteed. It would be possible to use this information, in conjunction with the `offsetof` macro to write code to access specific members of a structure, using pointers to other members. However, use of information on the relative ordering of structure members tends not to be code based, but data based (the same object is interpreted using different types). The coding guideline issues associated with the layout of types are discussed elsewhere.

1354 storage layout

1423 A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa.

pointer to structure points at initial member

Commentary

Although not worded as such, this is effectively a requirement on the implementation. The only reason for preventing implementations inserting padding at the start of a structure type is existing practice (and the resulting existing code that treats the address of a structure object as being equal to the address of the first member of that structure).

Other Languages

Most languages do not go into this level of representation detail.

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

569.1 representation information using

1424 There may be unnamed padding within a structure object, but not at its beginning.

structure unnamed padding

Commentary

Unnamed padding is needed when the next available free storage, for a member of a structure type, does not have the alignment required by the member type. Another reason for using unnamed padding is to mirror the layout algorithm used by another language, or even that used by another execution environment.

alignment 39
member 1421
alignment

The standard does not guarantee that two structure type having exactly the same member types have exactly the same storage layout, unless they are part of a common initial sequence.

structural 1585
compatibility
common initial sequence 1038

C90

There may therefore be unnamed padding within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

C++

This commentary applies to POD-struct types (9.2p17) in C++. Such types correspond to the structure types available in C.

Other Languages

No language requires implementations to pad members so that there is no padding between them. Few language specifications call out the fact that there may be padding within structure objects.

Common Implementations

Implementations usually only insert the minimum amount of unnamed padding needed to obtain the correct storage alignment for a member.

Coding Guidelines

The presence of unnamed padding increases the size of a structure object. Developers sometimes order members to minimize the amount of padding that is likely to be inserted by a translator. Ordering the members by size (either smallest to largest, or largest to smallest) is a common minimization technique. This is making use of layout information and a program may depend on the size of structure objects being less than a certain value (perhaps there may be insufficient available storage to be able to run a program if this limit is exceeded). However, it is not possible to tell the difference between members that have been intentionally ordered to minimize padding, rather than happening to have an ordering that minimizes (or gets close to minimizing) padding. Consequently these coding guidelines are silent on this issue.

Unnamed padding occupies storage bytes within an object. The pattern of bits set, or unset, within these bytes can be accessed explicitly by a conforming program (using `memcpy` or `memset` library functions). They may also accessed implicitly during assignment of structure objects. It is the values of these bytes that is a potential cause of unexpected behavior when the `memcmp` (amongst others) library function is used to compare two objects having structure type.

footnote 602
43

Example

```
#include <stdlib.h>

/*
 * In an implementation that requires objects to have an address that is a
 * multiple of their size, padding is likely to occur as commented.
 */
struct S_1 {
            char mem_1; /* Likely to be internal padding following this member. */
            long mem_2; /* Unlikely to be external padding following this member. */
           };
struct S_2 {
            long mem_1; /* Unlikely to be internal padding following this member. */
```

```
13              char mem_2; /* Likely to be external padding following this member. */
14             };
15
16  void f(void)
17  {
18  struct S_1 *p_s1 = malloc(4*sizeof(struct S_1));
19  struct S_2 *p_s2 = malloc(4*sizeof(struct S_2));
20  }
```

1425 The size of a union is sufficient to contain the largest of its members.

Commentary

A union may also contain unnamed padding.

1428 structure trailing padding

1426 The value of at most one of the members can be stored in a union object at any time.

union member at most one stored

Commentary

This statement is a consequence of the members all occupying overlapping storage and having their first byte start at the same address. The value of any bytes of the object representation that are not part of the value representation, of the member last assigned to, are unspecified.

531 union type overlapping members
1427 union members start same address
589 union member when written to

Other Languages

Pascal supports a construct, called a *variant tag*, that can be used by implementations to check that the member being read from was the last member assigned to. However, use of this construct does require that developers explicitly declare such a tag within the type definition. A few implementations perform the check suggested by the language standard. Ada supports a similar construct and implementations are required to perform execution time checks, when a member is accessed, on what it calls the *discriminant* (which holds information on the last member assigned to).

Common Implementations

The RTC tool[866] performs runtime type checking and is capable of detecting some accesses (it does not distinguish between different pointer types and different integer types having the same size) where the member read is different from the last member stored in.

1427 A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

union members start same address

Commentary

Although not worded as such, this is effectively a requirement on the implementation. A consequence of this requirement is that all members of a union type have the same offset from the start of the union, zero. A previous requirement dealt with pointer equality between different members of the same union object. This C sentence deals with pointer equality between a pointer to an object having the union type and a pointer to one of the members of such an object.

1207 pointer to union members compare equal

C++

This requirement can be deduced from:

> *Each data member is allocated as if it were the sole member of a* ***struct****.* 9.5p1

Other Languages

Strongly typed languages do not usually (Algol 68 does) provide a mechanism that returns the addresses of members of union (or structure) objects. The result of this C requirement (that all members have the same

address) are not always specified, or implemented, in other languages. It may be more efficient on some processors, for instance, for members to be aligned differently (given that in many languages unions may only be contained within structure declarations and so could follow other members of a structure).

Common Implementations

The fact that pointers to different types can refer to the same storage location, without the need for any form of explicit type conversion, is something that optimizers performing points-to analysis need to take into account.

Coding Guidelines

The issues involved in having pointers to different types pointing to the same storage locations is discussed elsewhere.

pointer 1299 qualified/unqualified versions

---

structure trailing padding

There may be unnamed padding at the end of a structure or union. 1428

Commentary

The reasons why an implementation may need to add this padding are the same as those for adding padding between members. When an array of structure or union types is declared, the first member of the second and subsequent elements needs to have the same alignment as that of the first element. In:

structure 1424 unnamed padding

```
union T {
          long m_1;
          char m_2[11];
         };
```

it is the alignment requirements of the member types, rather than their size, that determines whether there is any unnamed padding at the end of the union type. When one member has a type that often requires alignment on an even address and another member contains an odd number of bytes, it is likely that some unnamed padding will be used.

C++

The only time this possibility is mentioned in the C++ Standard is under the **`sizeof`** operator:

5.3.3p2 *When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array.*

Other Languages

The algorithms used to assign offsets to structure members are common to implementations of many languages, including the rationale for unnamed padding at the end. Few language definitions explicitly call out the fact that structure or union types may have unnamed padding at their end.

Common Implementations

Most implementations use the same algorithm for assigning member offsets and creating unnamed padding for all structure and union types in a program, even when these types are anonymous (performing the analysis to deduce whether the padding is actually required is not straight-forward). Such an implementation strategy is likely to waste a few bytes in some cases. But it has the advantage that, for a given implementation and set of translator options, the same structure declarations always have the same size (there may not be any standard's requirement for this statement to be true, but there is sometimes a developer expectation that it is true).

Coding Guidelines

Unnamed padding is a representation detail associated with storage layout. That this padding may occur after the last declared member is simply another surprise awaiting developers who try to make use of storage layout details. The guideline recommendation dealing with the use of representation information is applicable here.

storage 1354 layout representation information using 569.1

1429 As a special case, the last element of a structure with more than one named member may have an incomplete array type;

Commentary

The Committee introduced this special case, in C99, to provide a standard defined method of using what has become known as the *struct hack*. Developers sometimes want a structure object to contain an array object whose number of elements is decided during program execution. A standard, C90, well defined, technique is to have a member point at dynamically allocated storage. However, some developers, making use of representation information, caught onto the idea of simply declaring the last member be an array of one element. Storage for the entire structure object being dynamically allocated, with the storage allocation request including sufficient additional storage for the necessary extra array elements. Because array elements are contiguous and implementations are not required to perform runtime checks on array indexes, the additional storage could simply be treated as being additional array elements. This C90 usage causes problems for translators that perform sophisticated flow analysis, because the size of the object being accessed does not correspond to the size of the type used to perform the access. Should such translators play safe and treat all structure types containing a single element array as their last member as if they will be used in a *struct hack* manner?

The introduction of flexible array members, in C99, provides an explicit mechanism for developers to indicate to the translator that objects having such a type are likely to have been allocated storage to make use of the *struct hack*.

The presence of a member having an incomplete type does not cause the structure type that contains it to have an incomplete type.

C90

The issues involved in making use of the *struct hack* were raised in DR #051. The response pointed out declaring the member to be an array containing fewer elements and then allocating storage extra storage for additional elements was not strictly conforming. However, declaring the array to have a large number of elements and allocating storage for fewer elements was strictly conforming.

```
#include <stdlib.h>
#define HUGE_ARR 10000 /* Largest desired array. */

struct A {
          char x[HUGE_ARR];
         };

int main(void)
{
struct A *p = (struct A *)malloc(sizeof(struct A)
                                  - HUGE_ARR + 100); /* Want x[100] this time. */
p->x[5] = '?'; /* Is strictly conforming. */
return 0;
}
```

Support for the last member having an incomplete array type is new in C99.

C++

Support for the last member having an incomplete array type is new in C99 and is not available in C++.

Common Implementations

All known C90 implementations exhibit the expected behavior for uses of the *struct hack*. However, some static analysis tools issue a diagnostic on calls to `malloc` that request an amount of storage that is not consistent (e.g., smaller or not an exact multiple) with the size of the type pointed to by any explicit cast of its return value.

Coding Guidelines

Is the use of flexible arrays members more or less error prone than using any of the alternatives?

The *struct hack* is not widely used, or even widely known about by developers (although there may be some development communities that are familiar with it). It is likely that many developer will not be expecting this usage. Use of a member having a pointer type, with the pointed-to object being allocated during program execution, is a more common idiom (although more statements are needed to allocate and deallocate storage; and experience suggests that developers sometimes forget to free up the additional pointed-to storage, leading to storage leakage).

From the point of view of static analysis the appearance of a member having an incomplete type provides explicit notification of likely usage. While the appearance of a member having a completed array type is likely to be taken at face value. Without more information on developer usage, expectations, and kinds of mistakes made it is not possible to say anything more on these possible usages.

---

flexible array member

this is called a *flexible array member*. 1430

Commentary

This defines the term *flexible array member*.

C++

There is no equivalent construct in C++.

---

flexible array member ignored

~~With two exceptions~~ In most situations, the flexible array member is ignored. 1431

Commentary

The following are some situations where the member is ignored:

- forming part of a common initial sequence, even if it is the last member,
- compatibility checking across translation units, and
- if an initializer is given in a declaration (this is consistent with the idea that the usage for this type is to allocate variably sized objects via `malloc`).

---

structure size with flexible member

~~First, the size of the structure shall be equal to the offset of the last element of an otherwise identical structure that replaces the flexible array member with an array of unspecified length.[106)]~~ In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. 1432

Commentary

The C99 specification required implementations to put any padding before the flexible array member. However, several existing implementations (e.g., GNU C, Compaq C, and Sun C) put the padding after the flexible array member. Because of the efficiency gains that might be achieved by allowing implementations to put the padding after the flexible array member the committee decided to sanction this form of layout.

The wording was changed by the response to DR #282.

~~Second~~However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; 1433

Commentary

The structure object acts as if it effectively grows to fill the available space (but it cannot shrink to smaller than the storage required to hold all the other members).

1434 the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array.

Commentary

This is a requirement on the implementation. It effectively prevents an implementation inserting additional padding before the flexible array member, dependent on the size of the array. Fixing the offset of the flexible array member makes it possible for developers to calculate the amount of additional storage required to accommodate a given number of array elements.

1435 If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

Commentary

In the following example:

```
1  struct T {
2             int mem_1;
3             float mem_2[];
4            } *glob;
5
6  glob=malloc(sizeof(struct T) + 1);
```

insufficient storage has been allocated (assuming `sizeof(float) != 1`) for there to be more than zero elements in the array type of the member `mem_2`. However, the requirements in the C Standard are written on the assumption that it is not possible to create a zero sized object, hence this *as-if* specification.

Other Languages

Few languages support the declaration of object types requiring zero bytes of storage.

1436 EXAMPLE ~~Assuming that all array members are aligned the same, after the declarations:~~

EXAMPLE
flexible member

```
struct s { int n; double d[]; };
struct ss { int n; double d[1]; };
```

~~the three expressions:~~

```
sizeof (struct s)
offsetof(struct s, d)
offsetof(struct ss, d)
```

~~have the same value. The structure **structs** has a flexible array member **d**.~~
If **sizeof (double)** is 8, then after the following code is executed:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

and assuming that the calls to **malloc** succeed, the objects pointed to by **s1** and **s2** behave, for most purposes, as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

Following the further successful assignments:

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) +  6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]);        // valid
*dp = 42;                // valid
dp = &(s2->d[0]);        // valid
*dp = 42;                // undefined behavior
```

The assignment:

```
*s1 = *s2;
```

only copies the member **n** ; if any of the array elements are within the first **sizeof(structs)** bytes of the structure, these might be copied or simply overwritten with indeterminate values. ~~and not any of the array elements. Similarly:~~

```
struct s  t1 = { 0 };            // valid
struct s  t2 = { 2 };            // valid
struct ss tt = { 1, { 4.2 }};    // valid
struct s  t3 = { 1, { 4.2 }};    // invalid: there is nothing for the 4.2 to initialize

t1.n = 4;                        // valid
t1.d[0] = 4.2;                   // undefined behavior
```

Commentary

Flexible array members are a new concept for many developers and this extensive example provides a mini-tutorial on their use.

The wording was changed by the response to DR #282.

footnote 106

~~106) The length is unspecified to allow for the fact that implementations may give array members different alignments according to their lengths.~~ 1437

Commentary

One reason for an implementation to use different alignments for array members of different lengths is to take advantage of processor instructions that require arrays to be aligned on multiples of their length.

Motorola 39 56000

The wording was changed by the response to DR #282.

Forward references: tags (6.7.2.3). 1438

6.7.2.2 Enumeration specifiers

enumeration specifier syntax

1439

enum-specifier:
enum *identifier*$_{opt}$ **{** *enumerator-list* **}**
enum *identifier*$_{opt}$ **{** *enumerator-list* **, }**
enum *identifier*
enumerator-list:
enumerator
enumerator-list **,** *enumerator*
enumerator:
enumeration-constant
enumeration-constant **=** *constant-expression*

Commentary

Support for a trailing comma is intended to simplify the job of automatically generating C source.

C90

Support for a trailing comma at the end of an *enumerator-list* is new in C99.

C++

The form that omits the brace enclosed list of members is known as an elaborated type specifier, 7.1.5.3, in C++.

The C++ syntax, 7.2p1, does not permit a trailing comma.

Other Languages

Many languages do not use a keyword to denote an enumerated type, the type is implicit in the general declaration syntax. Those languages that support enumeration constants do not always allow an explicit value to be given to an enumeration constant. The value is specified by the language specification (invariably using the same algorithm as C, when no explicit values are provided).

Common Implementations

Support for enumeration constants was not included in the original K&R specification (support for this functionality was added during the early evolution of C[1180]). Many existing C90 implementations support a trailing comma at the end of an *enumerator-list*.

Coding Guidelines

A general discussion on enumeration types is given elsewhere. 517 enumeration set of named constants

The order in which enumeration constants are listed in an enumeration type declaration often follows some rule, for instance:

- *Application conventions* (e.g., colors of rainbow, kings of England, etc.).
- *Human conventions* (e.g., increasing size, direction— such as left-to-right, or clockwise, alphabetic order, etc.).
- *Numeric values* (e.g., baud rate, Roman numerals, numeric value of enumeration constant, etc.).

While ordering the enumeration constant definitions according to some rule may have advantages (directly mapping to a reader's existing knowledge or ordering expectations may reduce the effort needed for them to organize information for later recall), there may be more than one possible ordering, or it may not be possible to create a meaningful ordering. For this reason no guideline recommendation is made here. 0 developer expectations

Do the visual layout factors that apply to the declaration of objects also apply to enumeration constants? The following are some of the differences between the declarations of enumeration constants and objects: 1348.1 init-declarator one per source line

- There are generally significantly fewer declarations of enumerator constants than objects, in a program (which might rule out a guideline recommendation on the grounds of applying to a construct that rarely occurs in source).
- Enumeration constants are usually declared amongst other declarations at file scope (i.e., they are not visually close to statements). One consequence of this is that, based on declarations being read on as as-needed basis, the benefits of maximizing the amount of surrounding code that appears on the display at the same time are likely to be small. 770 reading kinds of

The following guideline recommendation is given for consistency with other layout recommendations.

Cg 1439.1 No more than one enumeration constant definition shall occur on each visible source code line.

The issue of enumeration constant naming conventions is discussed elsewhere. 792 enumeration constant naming conventions

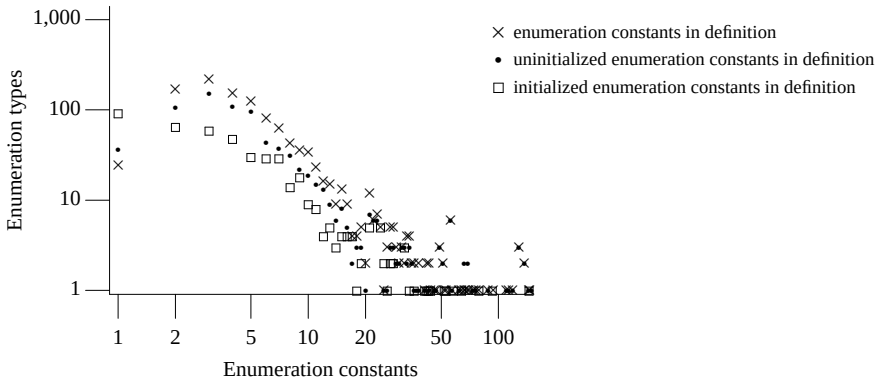


Figure 1439.1: Number of enumeration constants in an enumeration type and number whose value is explicitly or implicitly specified. Based on the translated form of this book's benchmark programs (also see Figure 298.1).

Usage

A study by Neamtiu, Foster, and Hicks[998] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 40% of releases one or more enumeration constants were added to an existing enumeration type while enumeration constants were deleted in 5% of releases and had one or more of their names changed in 16% of releases.[997]

Table 1439.1: Some properties of the set of values (the phrase *all values* refers to all the values in a particular enumeration definition) assigned to the enumeration constants in enumeration definitions. Based on the translated form of this book's benchmark programs.

Property	%
All value assigned implicitly	60.1
All values are bitwise distinct and zero is not used	8.6
One or more constants share the same value	2.9
All values are continuous , i.e. , number of enumeration constants equals maximum value minus minimum value plus 1	80.4

Constraints

enumeration constant representable in int

The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **int**. 1440

Commentary

constant 823 representable in its type
enumeration 864 constant type

This constraint is consistent with the requirement that the value of a constant be in the range of representable values for its type. Enumeration constants are defined to have type **int**.

C++

7.2p1 *The* `constant-expression` *shall be of integral or enumeration type.*

7.2p4 *If an initializer is specified for an enumerator, the initializing value has the same type as the expression.*

Source developed using a C++ translator may contain enumeration initialization values that would be a constraint violation if processed by a C translator.

```
#include <limits.h>

enum { umax_int = UINT_MAX}; /* constraint violation */
                             // has type unsigned int
```

Common Implementations

Some implementations support enumeration constants having values that are only representable in the types **unsigned int**, **long**, or **unsigned long**.

Coding Guidelines

The requirement is that the constant expression have a value that is representable as an **int**. The only requirement on its type is that it be an integer type. The constant expression may have a type other than **int** because of the use of a macro name that happens to have some other type, or because one of its operands happens to have a different type. If the constant expression consists, in the visible source, of an integer constant containing a suffix, it is possible that the original author or subsequent readers may assume some additional semantics are implied. However, such occurrences are rare and for this reason no guideline covering this case is given here.

There may be relationships between different enumeration constants in the same enumeration type. The issue of explicitly showing this relationship in the definition, using the names of those constants rather than purely numeric values, is a software engineering one and is not discussed further in these coding guidelines.

```
enum { E1 = 33, E2 = 36, E3 = 3 };

/* does not specify any relationship, and is not as resistant to modification as: */

enum { e1 = 33, e2 = e1+3, e3 = e2-e1 };
```

The enumeration constants defined in by an enumerated type are a set of identifiers that provide a method of naming members having a particular property. These properties are usually distinct and in many cases the values used to represent them are irrelevant.

Semantics

1441 The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.[107)]

enumerators type int

Commentary

The issues associated with enumeration constants having type **int** are discussed elsewhere, as are the issues of it appearing wherever such a type is permitted.

864 enumeration constant type
670 expression wherever an int may be used

C++

Following the closing brace of an `enum-specifier`*, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value.* 7.2p4

In C the type of an enumeration constant is always **int**, independently of the integer type that is compatible with its enumeration type.

```
#include <limits.h>

int might_be_cpp_translator(void)
{
enum { a = -1, b = UINT_MAX }; // each enumerator fits in int or unsigned int
```

```
6
7   return (sizeof(a) != sizeof(int));
8   }
9
10  void CPP_DR_172_OPEN(void) // Open C++ DR
11  {
12  enum { zero };
13
14  if (-1 < zero) /* always true */
15                 // might be false (because zero has an unsigned type)
16     ;
17  }
```

Other Languages

Most languages that contain enumerator types treat the associated enumerated constants as belonging to a unique type that is not compatible with type **int**. In these languages an enumeration constant must be explicitly cast (Pascal provides a built-in function, **ord**) before they can appear where a constant having type **int** may appear.

Coding Guidelines

object 1352 role
bit-set role 945

The values given to the enumeration constants in a particular enumeration type determine their role and the role of an object declared to have that type. To fulfil a bit-set role the values of the enumeration constants need to be bitwise distinct. All other cases create a type that has a symbolic role.

Example

```
1   enum T { attr_a = 0x01, attr_b = 0x02, attr_c = 0x04, attr_d = 0x10, attr_e = 0x20};
```

An enumerator with = defines its enumeration constant as the value of the constant expression. 1442

Commentary

This specifies the semantics associated with a token sequence permitted by the syntax (like the semantics of simple assignment, the identifier on the left of the **=** has as its value the constant expression on the right).

Other Languages

Not all languages that support enumeration constants allow the value, used to represent them during program execution, to be specified in their definition.

Coding Guidelines

macro 1931 object-like

Some guideline documents recommend against assigning an explicit value to an enumeration constant. Such recommendations limit enumeration types to having a symbolic role only. It has the effect of giving developers no choice but to use object-like macros to create sets of identifiers having bit-set roles. Using macros instead of enumerations makes it much more difficult for static analysis tools to deduce an association between identifiers (it may still be made apparent to human readers by grouping of macro definitions and appropriate commenting), which in turn will reduce their ability to flag suspicious use of such identifiers.

If the first enumerator has no =, the value of its enumeration constant is 0. 1443

Commentary

limit 298 enumeration constants

This choice is motivated by common usage and implementation details. Most enumeration types contain relatively few enumeration constants and many do not explicitly assign a value to any of them. Implicitly starting at zero means that it is possible to represent the values of all enumeration constants in an enumeration type within a byte of storage.

Other Languages

This is the common convention specified by other languages, or by implementations of other languages that do not specify the initial value.

1444 Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant.

Commentary

If the previous enumeration constant had the value MAX_INT, adding one will produce a value that cannot be represented in an **int**, violating a constraint.

1440 enumeration constant representable in int

Other Languages

This is the common convention specified by other languages, or by implementations of other languages that do not specify the initial value.

1445 (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.)

Commentary

When such enumeration constants are tested for equality with each other the result will be 1 (true), because it is their values not their spellings that are compared.

C++

The C++ Standard does not explicitly mention this possibility, although it does give an example, 7.2p2, of an enumeration type containing more than one enumeration constant having the same value.

Other Languages

No languages known to your author, that support the explicit definition of enumeration constant values, prohibits the appearance of duplicate values in the same enumeration.

Coding Guidelines

There are two ways in which more than one enumeration constant, in the same enumerated type, can have the same value. Either the values were explicitly assigned, or the at least one of the values was implicitly assigned its value. This usage may be an oversight, or it may be intentional (i.e., fixing the names of the first and last enumeration constant when it is known that new members may be added at a later date). These guideline recommendations are not intended to recommend against the creation of faults in code. What of the intended usage?

0 guidelines not faults

```
1   enum ET {FIRST_YEAR, Y_1898=FIRST_YEAR, Y_1899, Y_1900, LAST_KNOWN_YEAR=Y1900};
```

Do readers of the source assume there are no duplicate values among different enumeration constants, from the same enumerated type? Unfortunately use of enumerations constants are not sufficiently common among developers to provide the experience needed to answer this question.

1446 The enumerators of an enumeration are also known as its members.

Commentary

Developers often refer to the enumerators as *enumeration constants*, rather than *members*.

C++

The C++ Standard does not define this additional terminology for enumerators; probably because it is strongly associated with a different meaning for members of a class.

7.2p1

. . . the associated `enumerator` *the value indicated by the* `constant-expression`.

---

enumeration type compatible with

Each enumerated type shall be compatible with **`char`**, a signed integer type, or an unsigned integer type. 1447

Commentary

This is a requirement on the implementation. The term *integer types* cannot be used because enumerated types are included in its definition. There is no guarantee that when the **`sizeof`** operator is applied to an enumerator the value will equal that returned when **`sizeof`** is applied to an object declared to have the corresponding enumerator type.

integer types 519

C90

Each enumerated type shall be compatible with an integer type;

The integer types include the enumeration types. The change of wording in the C99 Standard removes a circularity in the specification.

integer types 519

C++

An enumeration is a distinct type (3.9.1) with named constants.

7.2p1

The underlying type of an enumeration may be an integral type that can represent all the enumerator values defined in the enumeration (7.2p5). But from the point of view of type compatibility it is a distinct type.

enumeration 864 constant type
enumeration 518 different type

It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than ***`int`*** *unless the value of an enumerator cannot fit in an* ***`int`*** *or* ***`unsigned int`***.

7.2p5

While it is possible that source developed using a C++ translator may select a different integer type than a particular C translator, there is no effective difference in behavior because different C translators may also select different types.

Other Languages

Most languages that support enumerated types treat such types as being unique types, that is not compatible with any other type.

Coding Guidelines

Experience shows that developers are often surprised by some behaviors that occur when a translator selects a type other than **`int`** for the compatible type. The two attributes that developers appear to assume an enumerated type to have are promoting to a signed type (rather than unsigned) and being able to represent all the values that type **`int`** can (if values other than those in the enumeration definition are assigned to the object).

If the following guideline recommendation on enumerated types being treated as not being compatible with any integer type is followed, these assumptions are harmless.

Experience with enumerated types in more strongly typed languages has shown that the diagnostics issued when objects having these types, or their members, are mismatched in operations with other types, are a very effective method locating faults. Also a number of static analysis tools[493, 683, 1157] perform checks on the use of objects having an enumerated type and their associated enumeration constants[1447.1] .

[1447.1] However, this is not necessarily evidence of a worthwhile benefit. Vendors do sometimes add features to a product because of a perceived rather actual benefit.

Cg 1447.1 Objects having an enumerated type shall not be treated as being compatible with any integer type.

Example

```
#include <stdio.h>

void f(void)
{
enum T {X};

if ((enum T)-1 < 0)
   printf("The type of enum {X} is signed\n");

if (sizeof(enum T) == sizeof(X))
   printf("The type of enum {X} occupies the same number of bytes as int\n");
}
```

1448 The choice of type is implementation-defined,[108)] but shall be capable of representing the values of all the members of the enumeration.

Commentary

This is a requirement on the implementation.

C90

The requirement that the type be capable of representing the values of all the members of the enumeration was added by the response to DR #071.

Other Languages

Languages that support enumeration types do not usually specify low level implementation details, such as the underlying representation.

Common Implementations

Most implementations chose the type **int**. A few implementations attempt to minimize the amount of storage occupied by each enumerated type. They do this by selecting the compatible type to be the integer type with the lowest rank, that can represent all constant values used in the definition of the contained enumeration constants.

Coding Guidelines

An implementation's choice of type will affect the amount of storage allocated to objects defined to have the enumerated type. The alignment of structure members having such types may also be affected. One reason why some developers do not use enumerated types is that they do not always have control over the amount of storage allocated. While this is a very minor consideration in most environments, in resource constrained environments it may be of greater importance.

A definition of an enumeration type may not include (most don't) enumeration constants for each of the possible values that can be represented in the underlying value representation (invariably some integer type). The guideline recommendation that both operands of a binary operator have the same enumerated type limits, but does not prevent, the possibility that a value not represented in the list of enumeration constants will be created.

517.3 enumeration constant as operand

Whether or not the creation of a value that is not represented in the list of enumeration constants is considered to be acceptable depends on the interpretation given to what *value* means. The approach taken by these coding guideline subsections is to address the issue from the point of view of the operators that might be expected to apply to the given enumeration type (these are discussed in the C sentence for the respective operators). The following example shows two possibilities:

```
enum roman_rep {
                 I = 1,
                 V = 5,
                 X = V+V,
                 L = V*X,
                 C = L+L,
                 D = C*V,
                 M = X*C
               } x;
enum termios_c_iflag {              /* A list of bitwise distinct values. */
                       BRKINT = 0x01,
                       ICRNL  = 0x02,
                       IGNBRK = 0x04,
                       IGNCR  = 0x10,
                       IGNPAR = 0x20,
                       INLCR  = 0x40
                      } y;

void f(void)
{
x= V+V;             /* Create a value whose arithmetic value is     represented. */
x= X+L;             /* Create a value whose arithmetic value is not represented. */
y= ICRNL | IGNPAR;  /* Create a value whose bit-set is     represented. */
y= ICRNL << 8;      /* Create a value whose bit-set is not represented. */
}
```

---

enumerated type incomplete until

The enumerated type is incomplete until after the `}` that terminates the list of enumerator declarations. 1449

Commentary

tag 1458 incomplete until

This sentence is a special case of one given elsewhere.

C90

The C90 Standard did not specify when an enumerated type was completed.

C++

The C++ Standard neither specifies that the enumerated type is incomplete at any point or that it becomes complete at any point.

7.2p4 *Following the closing brace of an* `enum-specifier`*, each enumerator has the type of its enumeration*

Example

The definition:

```
enum e_tag { e1 = sizeof(enum e_tag)};
```

is not permitted (it is not possible to take the size of an incomplete type). But:

```
enum e_tag { e1, e2} e_obj[sizeof(enum e_tag)];
```

is conforming.

EXAMPLE The following fragment: 1450

```
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
col = claret;
cp = & col;
if (*cp != burgundy)
        /* ... */
```

makes **hue** the tag of an enumeration, and then declares **col** as an object that has that type and **cp** as a pointer to an object that has that type. The enumerated values are in the set { 0, 1, 20, 21 }.

C++

The equivalent example in the C++ Standard uses the enumeration names red, yellow, green and blue.

Other Languages

In Pascal this example could be written as (no explicit assignment of values is supported):

```
1  type
2       hue : ( chartreuse, burgundy, claret, winedark );
```

and in Ada as:

```
1  type
2       hue is ( chartreuse, burgundy, claret, winedark );
3  for hue use (chartreuse => 0, burgundy => 1, claret => 20, winedark => 21);
```

1451 **Forward references:** tags (6.7.2.3).

1452 107) Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators. footnote 107

Commentary

This requirement can be deduced from the fact that enumeration constants are in the same name space as ordinary identifiers, they have no linkage, and that only one identifier with these attributes shall (a constraint) be declared in the same scope.

444 name space ordinary identifiers
432 identifier no linkage
1350 declaration only one if no linkage

C++

The C++ Standard does not explicitly make this observation.

Other Languages

Ada permits the same identifier to be defined as an enumeration constant in different enumerated type in the same scope. References to such identifiers have to be explicitly disambiguated.

1453 108) An implementation may delay the choice of which integer type until all enumeration constants have been seen. footnote 108

Commentary

This footnote is discussing the behavior of a translator that processes the source in a single pass. Many translators operate in a single pass and this behavior enables this form of implementation to continue to be used.

10 implementation single pass
10 implementation single pass

An enumerated type is incomplete until after the closing **}**, and there are restrictions on where an incomplete type can be used, based on the size of an object of that type. One situation where the size may be needed is in determining the representation used by an implementation for a pointer to a scalar type (there is no flexibility for pointers to structure and union types). An implementation does not know the minimum storage requirements needed to represent an object having an enumerated type until all of the members of that type had been processed. In the example below, a single pass implementation, that minimizes the storage allocated, and uses different representations for pointers to different scalar types, would not be able to evaluate `sizeof(enum e_T *)` at the point its value is needed to give a value to e2.

1449 enumerated type incomplete until
1465 footnote 109
559 pointer to qualified/unqualified types
560 alignment pointer to structures

```
1  struct s_T;
2  enum e_T {
```

```
3             e1=sizeof(struct s_T *),
4             e2=sizeof(enum e_T *),
5            };
6
```

C90

The C90 Standard did not make this observation about implementation behavior.

C++

This behavior is required of a C++ implementation because:

7.2p5 *The underlying type of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.*

Common Implementations

Some implementations unconditionally assign the type **int** to all enumerated types. Others assign the integer type with the lowest rank that can represent the values of all of the enumeration constants.

6.7.2.3 Tags

---

type contents defined once

A specific type shall have its content defined at most once. 1454

Commentary

declaration 1350 only one if no linkage

content 1462 list defines

The general requirement that an identifier with no linkage not be declared more than once does not apply to tags. An identifier denoting the same tag can be declared zero or more times if no content is defined. Among these declarations can be one that defines the content. What constitutes *content* is specified elsewhere.

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard does not classify the identifiers that occur after the **enum**, **struct**, or **union** keywords as tags. There is no tag namespace. The identifiers exist in the same namespace as object and typedef identifiers. This namespace does not support multiple definitions of the same name in the same scope (3.3p4). It is this C++ requirement that enforces the C one given above.

---

tag name same struct, union or enum

Where two declarations that use the same tag declare the same type, they shall both use the same choice of **struct**, **union**, or **enum**. 1455

Commentary

This sentence was added by the response to DR #251. The following code violates this constraint:

```
1  struct T {
2             int mem;
3            };
4  union T x; /* Constraint violation. */
```

C90

The C90 Standard did not explicitly specify this constraint. While the behavior was therefore undefined, it is unlikely that the behavior of any existing code will change when processed by a C99 translator (and no difference is flagged here).

C++

7.1.5.3p3

The `class-key` *or* **`enum`** *keyword present in the* `elaborated-type-specifier` *shall agree in kind with the declaration to which the name in the* `elaborated-type-specifier` *refers.*

Common Implementations

Early translators allowed the **`struct`** and **`union`** keywords to be intermixed (i.e., the above example was considered to be valid).

1456 A type specifier of the form

```
enum identifier
```

without an enumerator list shall only appear after the type it specifies is complete.

Commentary

Incomplete types are needed to support the declaration of mutually recursive structure and union types. It is not possible to create a mutually recursive enumerated type and a declaration making use of self-referencing recursion is an edge case that does not appear to be of practical use.

C90

This C99 requirement was not specified in C90, which did not containing any wording that ruled out the declaration of an incomplete enumerated type (and confirmed by the response to DR #118). Adding this constraint brings the behavior of enumeration types in line with that for structure and union types. 1118 sizeof constraints
Source code containing declarations of incomplete enumerator types will cause C99 translators to issue a diagnostic, where a C90 translator was not required to issue one.

```
1  enum E1 { ec_1 = sizeof (enum E1) };   /* Constraint violation in C99. */
2  enum E2 { ec_2 = sizeof (enum E2 *) }; /* Constraint violation in C99. */
```

C++

[Note: if the `elaborated-type-specifier` *designates an enumeration, the* `identifier` *must refer to an already declared* `enum-name`*.* 3.3.1p5

If the `elaborated-type-specifier` *refers to an* `enum-name` *and this lookup does not find a previously declared* `enum-name`*, the* `elaborated-type-specifier` *is ill-formed.* 3.4.4p2

```
1  enum incomplete_tag *x; /* constraint violation */
2                          // undefined behavior
3
4  enum also_incomplete; /* constraint violation */
5                        // ill-formed
```

Semantics

1457 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. tag declarations same scope

Commentary

What constitutes a declaration of structure, union, or enumerated type? The answer depends on whether a prior declaration, using the same identifier as a tag, is visible. If such an identifier is visible at the point of another declaration, no new type is declared (and there may also be a constraint violation).

struct-or-union identifier visible 1472
type contents defined once 1454

```
1  struct T_1 {   /* A declaration (1) of tag T_1. */
2              int mem_1;
3             };
4
5  void f(void)
6  {
7  struct T_1 *p; /* Prior declaration (1) is visible, not a declaration of T_1. */
8  struct T_1 {   /* A declaration (2) of tag T_1. */
9              float mem_2;
10            };
11 struct T_1 *q; /* Prior declaration (2) is visible and used. */
12 struct U_1 *r; /* No prior declaration of U_1 visible, a declaration of U_1. */
13 struct U_1 {   /* Defines the content of prior declaration of U_1. */
14             void *mem_3;
15            };
16
17 sizeof(p->mem_1);
18 sizeof(q->mem_2);
19 sizeof(r->mem_3);
20 }
```

struct-or-union identifier visible 1472

The wording that covers tags denoting the same type, but declared in different scopes occurs elsewhere.

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard specifies this behavior for class types (9.1p2). While this behavior is not specified for enumerated types, it is not possible to have multiple declarations of such types.

Coding Guidelines

identifier declared in one file 422.1

If the guideline recommendation specifying a single point of declaration is followed, the only situation where a tag, denoting the same type, is declared more than once is when its type refers to another type in some mutually recursive way.

---

tag incomplete until

The type is incomplete[109)] until the closing brace of the list defining the content, and complete thereafter. 1458

Commentary

The closing brace that defines its content may occur in a separate declaration. Incomplete types are one of the three kinds of types defined in C. The only other incomplete type is **void**, which can never be completed.

incomplete types 475
void is incomplete type 523

C++

The C++ Standard specifies this behavior for class types (9.2p2), but is silent on the topic for enumerated types.

---

tag declarations different scope

Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. 1459

Commentary

tag declarations same scope 1457

The discussion on what constitutes a declaration is also applicable here.

```
1  struct T_1 {           /* A declaration (1) of tag T_1. */
2              int mem_1;
3             };
4
5  void f(void)
6  {
7  struct T_1 *p; /* Prior declaration (1) is visible, not a declaration of T_1. */
8  struct T_1;    /* This is always a declaration, it is a different type from (1). */
9  struct T_1 *q; /* q points at a different type than p. */
10 }
11
12 void g(struct T_1 *); /* Prior declaration visible, not a declaration of T_1. */
13 void h(struct U_1 *); /* No prior declaration visible, a declaration of U_1.  */
```

C90

The C99 Standard more clearly specifies the intended behavior, which had to be inferred in the C90 Standard. 1457 tag declarations same scope

C++

The C++ Standard specifies this behavior for class definitions (9.1p1), but does not explicitly specify this behavior for declarations in different scope.

Coding Guidelines

If the guideline recommendation dealing with the reuse of identifier names is followed there will never be two distinct types with the same name. The case of distinct tags being declared with function prototype scope does not need a guideline recommendation. Such a declaration will render the function uncallable, as no type can be declared to be compatible with its parameter type. A translator will issue a diagnostic if a call to it occurs in the source. 792.3 identifier reusing names

---

1460 Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type. struct/union declaration no tag

Commentary

A declaration of a structure or union type that includes a tag may declare a distinct type, or it may refer to a previously declared distinct type.

If one of the identifiers declared is a typedef name, it will be possible to refer to the type in other contexts. If the identifier being declared is an object there is no standard defined way of referring to its type. Such types are sometimes known as *anonymous* types. 1468 footnote 110

Two types have compatible type if they are the same. Types that are distinct are not the same. 631 compatible type if

C90

The C90 Standard refers to a " . . . a new structure, union, or enumerated type," without specifying the distinctness of new types. The C99 Standard clarified the meaning.

C++

The C++ Standard specifies that a class definition introduces a new type, 9.1p1 (which by implication is distinct). However, it does not explicitly specify the status of the type that is created when the tag (a C term) is omitted in a declaration.

Other Languages

A small number of languages (e.g., CHILL) use structural equivalence for their type compatibility rules, rather than name equivalence. In such cases it is possible for many different type declarations to be treated as being compatible. 650 structural equivalence

Common Implementations

Some implementations (`gcc`) include the **`typeof`** operator. This returns the type of its operand. With the availability of such an operator no types can be said to be anonymous.

A type specifier of the form 1461

struct-or-union *identifier*$_{opt}$ **{** *struct-declaration-list* **}**

or

enum *identifier* **{** *enumerator-list* **}**

or

enum *identifier* **{** *enumerator-list* **,** **}**

declares a structure, union, or enumerated type.

Commentary

type specifier 1378 syntax struct tag; 1464

This specification provides semantics for a subset of the possible token sequences supported by the syntax of *type-specifier*. The difference between this brace delimited form and the semicolon terminated form is similar to the difference between the brace delimited and semicolon terminated form of function declarations (i.e., one specifies content and the other doesn't).

C90

Support for the comma terminated form of enumerated type declaration is new in C99.

C++

The C++ Standard does not explicitly specify this semantics (although 9p4 comes close).

content list defines

The list defines the *structure content*, *union content*, or *enumeration content*. 1462

Commentary

type 1454 contents defined once identifier 406 scope determined by declaration placement name space 438

This defines the terms *structure content*, *union content*, or *enumeration content*, which is the *content* referred to by the constraint requirement. The content is the members of the type declared, plus any type declarations contained within the declaration. Any identifiers declared by the list has the same scope as that of the tag, that might be defined, and may be in several name spaces.

Other Languages

Object-oriented languages allow additional members to be added to a class (structure) through the mechanism of inheritance.

tag declare

If an identifier is provided,[110)] the type specifier also declares the identifier to be the tag of that type. 1463

Commentary

This provides the semantic association for the identifier that appears at this point in the syntax.

C++

The term *tag* is not used in C++, which calls the equivalent construct a *class name*.

Table 1463.1: Occurrence of types declared with tag names (as a percentage of all occurrences of each keyword). Based on the visible form of the `.c` and `.h` files.

	`.c` files	`.h` files
union identifier	65.5	75.8
struct identifier	99.0	88.4
enum identifier	86.6	53.6

1464 A declaration of the form

struct tag;

```
struct-or-union identifier ;
```

specifies a structure or union type and declares the identifier as a tag of that type.[111]

Commentary

This form of declaration either declares, or redeclares, the identifier, as a tag, in the current scope. The following are some of the uses for this form of declaration:

- To support mutually referring declarations when there is the possibility that a declaration of one of the tags is already visible.
- To provide a mechanism for information hiding. Developers can declare a tag in an interface without specifying the details of a types implementation,
- In automatically generated code, where the generator does not yet have sufficient information to fully define the content of the type, but still needs to refer to it.

1474 EXAMPLE mutually referential structures

1465 109) An incomplete type may only by used when the size of an object of that type is not needed.

footnote 109 size needed

Commentary

When is the size of an object not needed? Who, or what needs the size and when do they need it?

The implementation needs the size of objects to allocate storage for them. When does storage need to be allocated for an object? In theory, not until the object is encountered during program execution (and in practice for a few languages). However, delaying storage allocation until program execution incurs a high-performance penalty. Knowing the size during translation enables much more efficient machine code to be generated. Also, knowing the size when the type is first encountered (if the size has to be known by the implementation) can simplify the job of writing a translator (many existing translators operated in a single pass).

1354 object reserve storage

10 implementation single pass

The size of an object having an incomplete array type is not needed to access an element of that array.

728 incomplete array indexing

The Committee responses to defect reports (e.g., DR #017) asking where the size of an object is needed do not provide a list of places. Now the wording has been moved to a footnote, perhaps this discussion will subside.

C90

It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.

The above sentence appears in the main body of the standard, not a footnote.

The C99 wording is more general in that it includes all incomplete types. This is not a difference in behavior because these types are already allowed to occur in the same context as an incomplete structure/union type.

475 incomplete types

C++

The C++ Standard contains no such rule, but enumerates the cases:

3.9p8

[Note: the rules for declarations and expressions describe in which contexts incomplete types are prohibited.]

Other Languages

Knowing the size of objects is an issue in all computer languages. When the size needs to be known is sometimes decided by high-level issues of language design (some languages require their translators to effectively perform more than one pass over the source code), other times it is decided by implementation techniques.

Common Implementations

implementation 10 single pass

Most C translators perform a single pass over the source code, from the point of view of syntactic and semantic processing. An optimizer may perform multiple passes over the internal representation of statements in a function, deciding how best to generate machine code for them.

---

size not needed examples

It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.) 1466

Commentary

typedef 1633 is synonym

alignment 560 pointer to structures

alignment 561 pointer to unions

A typedef name is simply a synonym for the type declared. All pointers to structure types and all pointers to union types have the same alignment requirements. No information on their content is required. The size may not be needed when a function returning a structure or union is declared, but it is needed when such a function is defined.

Other Languages

Many languages support some form of pointer to generic type, where little information (including size) is known about the pointed-to type. Support for type declarations where the size is unknown, in other contexts, varies between languages. In Java the number of elements in an array type is specified during program execution.

Common Implementations

This is one area where vendors are often silent on how their language extensions operate. For instance, the `gcc` **`typeof`** operator returns the type of its operand. However, the associated documentation says nothing about the case of the operand type being incomplete and having a tag that is identical to another definition occurring within the scope that the **`typeof`** occurred. One interpretation (unsupported by any specification from the vendor) of the following:

```
1  extern struct fred f;
2
3  int main (void)
4  {
5  typeof (f) *x;
6  struct fred { int x; } s;
7  typeof (f) *y;
8  y=&s;                  /* Types not compatible? */
9  }
10
11 struct fred {
12              int mem;
13             };
```

is that both `x` and `y` are being declared as being pointers to the type of `f`, that is an incomplete type, and that the declaration of the tag `fred`, in a nested scope, has no effect on the declaration of `y`.

Example

```
1  typedef struct foo T;
2
3  struct foo *ptr;
4  struct foo f(void);
5
6  void g(void *p)
7  {
8  (struct foo *)p;
9  }
```

1467 The specification has to be complete before such a function is called or defined.

Commentary

In these contexts the commonly used methods for mapping source code to machine code need to know the number of bytes in a types object representation.

C90

The specification shall be complete before such a function is called or defined.

The form of wording has been changed from appearing to be a requirement (which would not be normative in a footnote) to being commentary.

1468 110) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part.

footnote 110

Commentary

The requirements for referring to objects declared using such types are discussed elsewhere. Such types are distinct and are said to be *anonymous*. They cannot be referred to elsewhere in the translation unit. Although their associated objects can be accessed. In:

633 compatible separate translation units
1460 struct/union declaration no tag

```
1  struct {
2           int m1;
3          } x, y;
4  struct {
5           int m1;
6          } z;
```

x and y are compatible with each other. They both have the same anonymous type, but the object z has a different anonymous type. Note that the types of the objects x, y, and z would be considered to be compatible if they occurred in different translation units.

633 compatible separate translation units

C90

This observation was is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

Other Languages

Some languages include a **`typeof`** operator, which returns the type of its operand.

Common Implementations

Some implementations include a **typeof** operator. This returns the type of its operand. The availability of such an operator means that no types never need be truely anonymous.

Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type. 1469

Commentary

Use of a typedef name does not alter the requirements on the type not being an incomplete type in some contexts. size needed 1465

```
1  struct S;
2  typedef struct S t_1;
3  extern t_1 f(void);
4
5  struct S {
6            int mem;
7           };
8  typedef struct S t_2;
9  t_2 glob;
10
11 t_1 f(void)
12 {
13 return glob;
14 }
```

C90

This observation is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

Example

In the following both a and b have the same type. The typedef name T_S provides a method of referring to the anonymous structure type.

```
1  typedef struct {
2                 int m2;
3                 } T_S;
4  T_S a;
5  T_S b;
```

---

111) A similar construction with **enum** does not exist. footnote 111 1470

Commentary

The need for mutual recursion between different enumerated types is almost unheard of. One possible use of such a construct might be to support the hiding of enumeration values. For instance, an object of such an enumeration type might be passed as a parameter which only ever appeared as an argument to function calls. However, C99 considers the following usage to contain a number of constraint violations.

```
1  enum SOME_STATUS_SET; /* First constraint violation. */
2
3  extern enum SOME_STATUS_SET get_status(void);
4  extern void use_status(enum SOME_STATUS_SET *);
```

C++

If an `elaborated-type-specifier` *is the sole constituent of a declaration, the declaration is ill-formed unless . . .* 7.1.5.3p1

The C++ Standard does not list `enum identifier ;` among the list of exceptions and a conforming C++ translator is required to issue a diagnostic for any instances of this usage.

The C++ Standard agrees with this footnote for its second reference in the C90 Standard.

1471 struct-or-union identifier not visible

1471 If a type specifier of the form

struct-or-union identifier not visible

struct-or-union identifier

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[111]

Commentary

The forms of *struct-or-union identifier*, excluded by this wording, are the identifier being followed by a semicolon or a left brace. The remaining possible occurrences of this form are described elsewhere and include:

1466 size not needed examples

```
1  struct secret_info *point_but_not_look;
```

C++

The C++ Standard does not explicitly discuss this kind of construction/occurrence, although 3.9p6 and 3.9p7 discuss this form of incomplete type.

Coding Guidelines

When no other declaration is visible at the point this type specifier occurs, should this usage be permitted? Perhaps it was intended that a tag be visible at the point in the source where this type specifier occurs. However, not having a prior declaration visible is either harmless (intended or otherwise), or will cause a diagnostic to be issued by a translator.

A pointer to an incomplete structure or union type is a more strongly typed form of generic pointer than a pointer to **void**. Whether this use of pointer to incomplete types, for information hiding purposes, is worthwhile can only be decided by the developer.

1472 If a type specifier of the form

struct-or-union identifier visible

struct-or-union identifier

or

enum *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

Commentary

The forms of *struct-or-union identifier*, excluded by this wording, are the identifier being followed by a semicolon or a left brace. This is the one form that is not a declaration.

Coding Guidelines

type 1454 contents defined once

culture of C 0

Technically the only reason for using tags is to define mutually recursive structure or union types. However, in practice this is the most common form used to declare objects having structure, union, or enumerated types. In general these coding guidelines recommend that developers continue to following common existing practices. Given that most existing code contains declarations that use this form of type specifier, is there a worthwhile benefit in recommending use of an alternative form? The following are some of the issues involved in the obvious alternative form, the use of typedef names:

- While the use of a typedef name may appear to reduce future maintenance costs (e.g., if the underlying type changes from a structure to an array type, a single edit to the definition of a typedef name is sufficient to change to any associated object declarations). In practice the bulk of the costs associated with such a change are created by the need to modify the operators used to access the object (i.e., from a member selection operator to a subscript operator). Also experience suggests that this kind of change in type is not common.
- Changes in an objects structure type may occur as a program evolves. For instance, the object `x` may have structure type `t_1` because it needs to represent information denoted by a few of the members of that type. At a later time the type `t_1` may be subdivided into several structure types, with the members referenced by `x` being declared in the type `t_1_3`. Developers then have the choice of changing the declaration of `x` to be `t_1_3`, or leaving it alone. However, the prior use of a typedef name, rather than a tag, is unlikely to result in any cost savings, when changing the declaration of `x` (i.e., developers are likely to have declared `x` to have type `t_1`, rather than a synonym of that type, so the declaration of `x` will either have to be edited).
- What are the cognitive costs and benefits associated with the presence, or absence of a keyword in the source of a declaration? There is a cost to readers in having to process an extra token (i.e., the keyword) in the visible source, or any benefits, to readers of the visible source. However, the visual presence of this keyword may reduce the cognitive effort needed to deduce the kind of declaration being made. There does not appear to be a significant cost/benefit difference between any of these cognitive issues.

EXAMPLE 1 This mechanism allows declaration of a self-referential structure. 1473

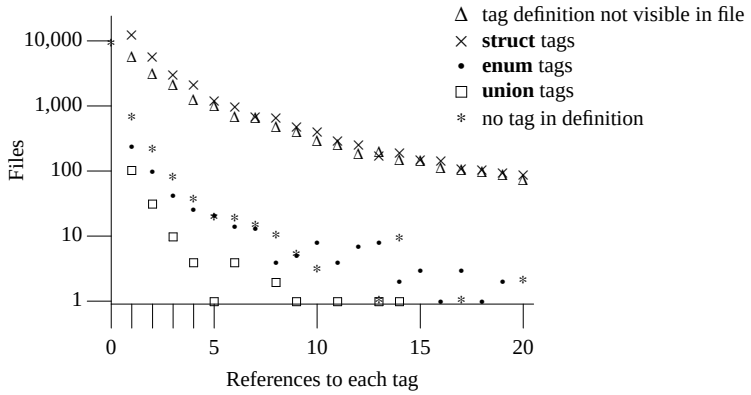


Figure 1472.1: Number of files containing a given number of references to each tag previously defined in the visible source of that file (times, bullet, square; the definition itself is not included in the count), tags with no definition visible in the `.c` file (triangle; i.e., it is defined in a header) and anonymous structure/union/enumeration definitions (star). Based on the visible form of the `.c` files.

```
struct tnode {
        int count;
        struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be an object of the given type and **sp** to be a pointer to an object of the given type. With these declarations, the expression **sp->left** refers to the left **struct tnode** pointer of the object to which **sp** points; the expression **s.right->count** designates the **count** member of the right **struct tnode** pointed to from **s**. The following alternative formulation uses the **typedef** mechanism:

```
typedef struct tnode TNODE;
struct tnode {
        int count;
        TNODE *left, *right;
};
TNODE s, *sp;
```

Commentary

Both of these formulations of commonly seen in source code.

Other Languages

Creating a self-referential type in Pascal requires the definition of two type names.

```
1  type
2     TNODE_PTR = ^TNODE; (* special rules covers this forward reference case *)
3     TNODE = record
4             count : integer;
5             left,
6             right : TNODE_PTR
7             end;
```

1474 EXAMPLE 2 To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

EXAMPLE mutually referential structures

```
struct s1 { struct s2 *s2p; /* ... */ }; // D1
struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

specify a pair of structures that contain pointers to each other. Note, however, that if **s2** were already declared as a tag in an enclosing scope, the declaration **D1** would refer to *it*, not to the tag **s2** declared in **D2**. To eliminate this context sensitivity, the declaration

```
struct s2;
```

may be inserted ahead of **D1**. This declares a new tag **s2** in the inner scope; the declaration **D2** then completes the specification of the new type.

Commentary

```
1  struct s2 { int mem1; };
2
3  void f(void)
4  {
5  struct s2;
6  struct s1 { struct s2 *s2p; /* ... */ };
7  struct s2 { struct s1 *s1p; /* ... */ };
8  }
```

without the declaration of the tag s2 in the body of f, the declaration at file scope would be visible and the member s2p would refer to it, rather than the subsequence definition in the same scope,

C++

This form of declaration would not have the desired affect in C++ because the braces form a scope. The declaration of s2 would need to be completed within that scope, unless there was a prior visible declaration it could refer to.

Forward references: declarators (6.7.5), array declarators (6.7.5.2), type definitions (6.7.7). 1475

6.7.3 Type qualifiers

type qualifier syntax

1476

type-qualifier:
const
restrict
volatile

Commentary

Rationale

Type qualifiers were introduced in part to provide greater control over optimization. Several important optimization techniques are based on the principle of "cacheing": under certain circumstances the compiler can remember the last value accessed (read or written) from a location, and use this retained value the next time that location is read. (The memory, or "cache", is typically a hardware register.) If this memory is a machine register, for instance, the code can be smaller and faster using the register rather than accessing external memory.

C90

Support for **restrict** is new in C99.

C++

Support for **restrict** is new in C99 and is not specified in the C++ Standard.

Other Languages

Some languages use the keyword **read**, or **readonly** as a type qualifier to indicate that an object can only be read from. BCPL uses **MANIFEST**.

Common Implementations

The keyword **noalias** was included in some drafts of the C90 Standard. It provided functionality whose intended use was similar to that provided by the keyword **restrict** in C99.

Coding Guidelines

declaration specifier ordering 1357.1

A guideline on the relative order of type qualifiers within a declaration specifier is given elsewhere.

Usage

Developers do not always make full use of the **const** qualifier. An automated analysis[437] of programs whose declarations contained a relatively high percentage (29%) of **const** qualifiers found that it would have been possible to declare 70% of the declarations using this qualifier. Engblom[389] reported that for real-time embedded C code 17% of object declarations contained the **const** type qualifier.

Table 1476.1: Common token sequences containing *type-qualifier*s (as a percentage of each *type-qualifier*). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
; volatile	0.1	36.1	**{ const**	0.2	5.6
, **const**	0.2	32.8	**const unsigned**	6.2	1.4
(const	0.2	28.1	**const struct**	11.1	1.3
(volatile	0.0	26.2	**volatile unsigned**	25.6	1.1
; const	0.1	14.1	**const void**	5.3	0.8
identifier **volatile**	0.0	11.4	**volatile struct**	15.5	0.4
{ volatile	0.1	11.0	**volatile int**	7.4	0.1
const char	54.1	10.4	**volatile** identifier	36.2	0.0
static const	1.5	10.0	**volatile (**	8.9	0.0
static volatile	0.3	8.6	**const** identifier	17.6	0.0

Constraints

1477 Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

restrict constraint

Commentary

The specification of the **restrict** qualifier only defines the behavior for pointers that refer to objects (because it is only intended to deal with such quantities). In the case of function parameters having array type (e.g., `void f(float a[restrict][100]);`) the implicit conversion to pointer type occurs before this constraint is applied (this intent is expressed in WG14/N521, an example in the standard, and the rationale).

1502 restrict formal definition
729 array converted to pointer
1622 EXAMPLE compatible function prototypes
1599 function declarator static

Semantics

1478 The properties associated with qualified types are meaningful only for expressions that are lvalues.[112)]

qualifier meaningful for lvalues

Commentary

Although type qualifiers are specified in terms of creating new type, they really act as modifiers of the declarator. Use of these keywords gives a type additional properties. However, they do not change its representation or alignment requirements. These properties are associated with the declared object, not its value (although they all specify something about the possible attributes of the values by the objects declared using them). A qualified type can be applied to a non-lvalue through the use of a cast operator (e.g., `(const long)1`).

556 qualifiers representation and alignment

C++

The C++ Standard also associates properties of qualified types with rvalues (3.10p3). Such cases apply to constructs that are C++ specific and not available in C.

Common Implementations

Some implementations support directives that allow the developer to specify which areas of storage objects or literals are to be held in.

Coding Guidelines

The fact that type qualifiers are only meaningful for lvalue expressions does not prevent developers using them in other contexts. Such usage is redundant and does not affect how a translator should interpret the behavior of a program (the issue of redundant code is discussed elsewhere). Is there a worthwhile benefit recommending against the use of type qualifiers in contexts where they have no meaning? The usage, in the visible source and not via a typedef name, suggests that the author may believe it has some affect on the behavior of the program. In practice such usage is very rare and the developer misconception harmless.

190 redundant code

1479 If the same qualifier appears more than once in the same *specifier-qualifier-list*, either directly or via one or more **typedef**s, the behavior is the same as if it appeared only once.

qualifier appears more than once

Commentary

Having the same qualifier appear more than once in the same *specifier-qualifier-list* may be redundant, but it is harmless. Support for such usage can simplify the automatic generation of C source code and reduce the amount of coordination needed between different development groups (e.g., over who is responsible for ensuring a given qualifier appears in a chain of typedef names; qualifiers can appear on in typedef name defined by a group, independent of any references it makes to typedef names defined by other development groups). It can also reduce maintenance costs for existing source (e.g., a change to the definition of a typedef name does not have any cascading affects on any existing declarations it appears in with qualifiers). The same permission is given for function specifiers.

function specifier appears more than once 1527

C90

The following occurs within a Constraints clause.

> *The same type qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more* **typedef***s.*

Source code containing a declaration with the same qualifier appearing more than once in the same *specifier-qualifier-list* will cause a C90 translator to issue a diagnostic.

C++

7.1.5p1 > *However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3) or template type arguments (14.3), in which case the redundant cv-qualifiers are ignored.*

The C++ Standard does not define the term *prohibited*. Applying common usage to this term suggests that it is to be interpreted as a violation of a diagnosable (because "no diagnostic is required", 1.4p1, has not been specified) rule.

The C++ specification is intermediate between that of C90 and C99.

```
1  const const int cci; /* does not change the conformance status of program */
2                       // in violation of a diagnosable rule
```

Coding Guidelines

Is there a worthwhile benefit recommending against having the same qualifier appears more than once in the same *specifier-qualifier-list*?

There is an obvious organizational and maintenance benefit in allowing the same qualifier to occur more than once via typedefs. Having the same qualifier appear more than once in the visible source of a *specifier-qualifier-list* suggests that insufficient attention was invested by the original author. This usage also requires subsequent readers to invest more cognitive effort in comprehending the declaration. However, support for this usage is new in C99 and measurements of source show an underuse of the most common qualifier (i.e., **const**). Overuse of qualifiers does not look like it will be an issue that needs addressing via a guideline. Such usage is redundant and does not affect the behavior of a program (the issue of redundant code is discussed elsewhere).

redundant code 190

Example

```
1  const const int glob_1;
2  const int const long const extern const signed const glob_2;
3
4  typedef const int C_I;
5  typedef const C_I const_C_I;
```

1480 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined.

const qualified attempt modify

Commentary

Modifying a const-qualified object requires the use of a pointer to it (plus an appropriate cast operation; attempting to modify the object using its declared type is a constraint violation). The undecidability of detecting all such pointer usages at translation time and the overhead of performing the check during program execution resulted in the committee specifying the behavior as undefined, rather than a constraint violation.

1289 assignment operator modifiable lvalue

1502 restrict formal definition

The author of the source may intend a use of the **const** qualifier to imply a read-only object, and a translator may treat it as such. However, a translator is not required to perform any checks at translation or execution time to ensure that the object is not modified (via some pointer to it).

C++

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [Example: a member function called for an object (9.3) can modify the object.]

3.10p10

The C++ Standard specifies a different linkage for some objects declared using a const-qualified type.

425 static internal linkage

Other Languages

Languages that contain some form of readonly qualifier usually specify that so qualified objects cannot be modified and attempts to perform such modifications are treated as errors. There is usually some indirect mechanism that will have the effect of modifying such objects, and few implementations are required to detect such modifications during program execution.

Common Implementations

Some freestanding implementations place static storage duration, const-qualified, objects in read-only memory. While it might be possible to write to this memory, the value of objects held there are not modified by such operations. Some hosted implementations place static storage duration, const-qualified, objects in a region of storage marked as read-only (some processor memory management units support such types of memory and automatically perform checks on accesses to it).

Many implementations place const-qualified objects in the same kind of storage as other objects. However, this does not mean that if these qualified objects are modified, the modified value is actually used by a program. A translator may reduce optimization overhead by assuming const-qualified objects are never modified. This could result in values held in registers being reused, after the object held in storage had been modified.

Coding Guidelines

There are many different ways of attempting to modify a const-qualified object. Enumerating all such cases would create an unnecessarily large number of guidelines, and is not guaranteed to catch all cases. To be able to attempt to modify a const-qualified object, without a translator issuing a diagnostic, it is necessary to use an explicit cast. A single guideline covering this case is discussed elsewhere and deals with the issue at the point potential for problems starts.

1135 pointer conversion constraints

Example

```
1  extern const int g_1 = 3;
2  extern int g_2;
3
4  void f(const int *pi);
5  /*
```

```
 * The declaration of pi implies that the statements within f will
 * not modify its value.  However, an optimizer cannot assume that
 * the value *pi will remain unchanged throughout the execution of
 * f.  For instance the function f may modify g_2; whose address
 * is passed as an argument in the second call below.
 */

void g(void)
{
const int *loc_1 = &g_1;
int *loc_2 = &g_2;

f(loc_1);
f((const int *)loc_2);
}
```

EXAMPLE 1309 const pointer

Also see an example elsewhere.

---

volatile qualified attempt modify

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.[113)] 1481

Commentary

const 1480 qualified attempt modify

The issues are the same as for the const-qualified case.

Common Implementations

The extent to which the behavior, in this case, differs from that intended will depend on the optimizations performed by an implementation. If an implementation has reused a value, held in a register because it was accessed via a non-volatile-qualified type, there will be no access to the volatile-qualified object. Presumably the object was declared to have volatile-qualified type because accesses to it caused some external affect. The difference on the behavior of a program, because of less accesses to such a qualified object can only be guessed at.

Coding Guidelines

pointer conversion 1135 constraints

Like the const-qualified case, the appropriate guideline recommendation is discussed elsewhere.

Example

```
extern volatile int glob_1;

void f(int *p_1)
{
(*p_1)++;
(*p_1)++;
}

void g(void)
{
f((int *)glob_1);
}
```

---

volatile last-stored value

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. 1482

Commentary

The **volatile** qualifier informs the implementation that it would be unsafe to optimize accesses to objects declared using it.

Other Languages

The **volatile** field modifier in Java is used to provide a method of ensuring that accesses to objects whose values may be modified by different threads are reconciled with the master copy in shared memory.

Common Implementations

The **volatile** qualifier is mainly seen in freestanding implementations and hosted implementations where storage is shared between multiple programs. The following are some of the behaviors often seen for objects declared using this qualifier:

- Updating an object on a periodic basis (e.g., a realtime clock).
- Modifying an object after every read from it (e.g., instance, an input device sending characters down a serial line).
- An object may not contain the value last written to it (e.g., an output port, where reading from that location returns the status of the write operation).

The only meaningful volatile-qualified objects are often declared by the implementation, as part of a vendor supplied API. This is because having an object modified in ways unknown to the implementation usually involves associating it with some external device. Such associations usually require making use of some implementation provided language extension, or by passing the address of the object to a vendor supplied library function.

Coding Guidelines

Qualifying an object declaration with **volatile** may inform the translator that it may be modified in ways unknown, but how well will developers understand the implications of the changing value? For instance, in:

```
1  extern volatile long seconds_since_midnight;
2
3  void f(void)
4  {
5  int now_hours = (seconds_since_midnight / 3600);
6  int now_minutes = (seconds_since_midnight / 60);
7  int now_seconds = (seconds_since_midnight % 60);
8  }
```

the time at the start of the evaluation of the expression assigned to `now_hours` may have been 9:59:59 and at the start of the evaluation of the expression assigned to `now_minutes` 10:00:00. The values of the three objects would then be identical to the time obtained at 9:00:00.

Rev 1482.1 A sequence of related expressions that accesses the value of the same volatile-qualified object shall be checked to ensure the changeability of the volatile nature of the objects value has been taken into account.

1483 Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3.

Commentary

This is a requirement on the implementation. The possibility of a third-party library configuring a hardware device to update the contents of an object, whose address was passed as a parameter, means that implementations cannot assume that volatile-qualified objects contain known values, just because the implementation provides no mechanism for associating them with a hardware device.

C++

There is no equivalent statement in the C++ Standard. But it can be deduced from the following two paragraphs:

1.9p5 *A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input.*

1.9p6 *The observable behavior of the abstract machine is its sequence of reads and writes to* **`volatile`** *data and calls to library I/O functions.*[6)]

Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[114)] 1484

Commentary

This is a requirement on the implementation. As well as performing the same number of accesses required by the abstract machine, updates to the stored value must have occurred in the order in which sequence points are reached (multiple accesses to the same volatile-qualified object between successive sequence points results in undefined behavior). The extent to which this requirement gives developers the ability to predict the value last stored in an object will depend on the uniqueness of the ordering of sequence points. These issues are also discussed in a C Standard example given elsewhere.

object 941 modified once between sequence points
sequence 187 points
abstract 206 machine
example

---

footnote 112

112) The implementation may place a **`const`** object that is not **`volatile`** in a read-only region of storage. 1485

Commentary

The following are some of possible benefits of using read-only storage:

- the cost of ROM is significantly lower than that of RAM,
- the contents of ROM are not lost when power to the computer is switched off, and
- it may be possible to cause write accesses to regions of storage that an operating system has marked as being read-only to raise an exception that can be caught by a program debugging tool.

const volatile 1496 EXAMPLE

In some cases a **`const`** object that is **`volatile`** can be in a read-only region of storage.

C++

The C++ Standard does not make this observation.

Common Implementations

Sometimes the storage is physically read-only, as in ROM hardware. Sometimes implementations make use of a host operating system's ability to configure regions of storage as read-only, after the program image has been loaded, and enforcing that requirement (invariably through hardware assisted, memory management support).

Moreover, the implementation need not allocate storage for such an object if its address is never used. 1486

Commentary

An implementation need not allocate storage for any object whose address is never used. The difference with const-qualified objects is that a translator often knows in advance what their value is going to be, through out the execution of a program (it appears in the initialization of the objects definition). If this value is a constant expression the translator can substitute it wherever the identifier denoting the const-qualified object occurs. A translator could treat members of a const-qualified structure object in a similar fashion. If such substitutions were made, the undefined behavior associated with any attempts to modify the value being to have no effect.

1480 const qualified attempt modify

C++

The C++ Standard does not make this observation. However, given C++ supports zero sized objects, 1.8p5, there may be other cases where implementations need not allocate storage.

Common Implementations

Replacing references to objects by their known values, at a given point in a program, is not limited to const-qualified objects. However, significantly more analysis is needed for translators to perform this kind of substitution on references to non-const-qualified objects.

1487 113) This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

footnote 113

Commentary

Such objects may exist in the addressable storage of a program and accessed by casting an integer value, representing that address, to a pointer type. The properties associated with the qualification of a pointed-to type always apply, no matter what location in storage is pointed at.

C++

The C++ Standard does not make this observation.

1488 What constitutes an access to an object that has volatile-qualified type is implementation-defined.

Commentary

This implementation-defined behavior is caused by the different possible ways an implementation can access an objects having bit-field types.

```
1   extern volatile int glob;
2
3   struct {
4           volatile unsigned int m_1 :5;
5                    unsigned int m_2 :5;
6           volatile unsigned int m_3 :5;
7          } x;
8
9   void f(void)
10  {
11  glob = glob + 1;
12  glob++;
13
14  x.m_1 = 2;
15  x.m_2 = 3;
16  int loc = x.m_2 + x.m_2;
17  }
```

The first increment of `glob` would generally be considered to involve two accesses. The second increment of `glob` could be performed in one access on some processors (those whose having an increment memory instruction), but two accesses on other processors (the majority, which effectively perform the sequence load/increment/store).

In the second set of assignments the access methods are likely to be fundamentally different. One or more bit-fields might be packed into the same storage unit. The assignment to `m_1` may need to read from the storage unit that holds its value, to obtain the value of the bits representing the member `m_2` (so they can be bitwise-AND'ed with the new value, unless the processor supports some form of bit-field access instruction). Assignments to non-bit-field objects do not normally involve a read of their value. An even more surprising access could occur through the assignment to `m_2`, which could also cause the members `m_1` or `m_3` to be accessed, even although they are not explicitly mentioned in the assignment.

bit-field 1410 packed into

bit-field 672 in expression

Volatile-qualified objects can also be affected by translator optimizations. The evaluation of the expression assigned to `loc` could cause the value of `x.m_2` to be held in a register, so it would not need to be loaded for the second access. This optimization would potentially result in one fewer accesses to `x.m_1`.

C++

The C++ Standard does not explicitly specify any behavior.

7.1.5.1p8 *[Note: . . . In general, the semantics of* **`volatile`** *are intended to be the same in C++ as they are in C.]*

Common Implementations

In those implementations that support volatile-qualified objects whose value is modified in ways unknown to the implementation, the objects have non-bit-field scalar types.

Coding Guidelines

This implementation-defined behavior has the ability to generate many surprises for developers. However, implementations that support volatile-qualified objects whose value is modified in ways unknown to the implementation and whose object representation only occupies part of a storage unit are very rare. For this reason no guideline recommendations are made here.

An object that is accessed through a restrict-qualified pointer has a special association with that pointer. 1489

Commentary

The emerging common terminology usage is the term *restricted pointer* (which is also used in the standard) rather than *restrict-qualified pointer*.

It is not possible to declare a restrict-qualified object, so all restrict-qualified pointers are created by the conversion of a non-restrict-qualified address. The equivalent association for objects accessed via their declared name can be obtained by using the **`register`** storage class in the objects declaration. Because it is not possible to take the address of such an object it cannot have any aliases.

restrict 1477 constraint

unary & 1088 operand constraints

As the specification of what an object is composed of, and an example given in the standard show, the object accessed through a restrict-qualified pointer could be part of a larger object.

object 570 contiguous sequence of bytes

restrict 1519 example 2

C90

Support for the **`restrict`** qualifier is new in C99.

C++

Support for the **`restrict`** qualifier is new in C99 and is not available in C++.

Other Languages

Optimizing the performance of identical operations on array objects is one of the intended uses of restrict-qualified pointers. Some languages (Fortran 90, Ada) achieve the same result by supporting operations that can be applied to every element in an array, or slice of an array.

This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.[115)] 1490

restrict requires all accesses

Commentary

This association only applies within the lifetime of the restricted pointer object, which may be a subset of the lifetime of the pointed-to object. It enables optimizations to be localized to accesses of an object within a particular scope (e.g., a function definition or loop). The optimization being driven by the method used to access the object, rather than the object itself.

451 lifetime of object
1503 restrict pointer lifetime

There is no requirement on translators to check that a particular restricted pointer is the only method used to access the pointed-to object. Such a requirement would be counter productive, since the purpose of this qualifier is to overcome the technical difficulties associated with deducing the information it denotes automatically (although algorithms for automatically deducing some cases are known[10]). If this requirement is not met the behavior is undefined. A translator that relies on the presence of the **restrict** qualifier, to perform optimizations, may produce different results in this case, than if the qualifier did not appear in the pointer declaration.

1513 restrict undefined behavior

This association only applies if the access is through a restricted pointer. Assigning the value of a restricted pointer to a non-restricted pointer does not cause the special association to also be assigned. For instance, in the following example, accesses through the pointers p and q are not restrict-qualified.

```
void f(float * restrict r,
       float * restrict s,
       int len)
{
float *p = r,
      *q = s;

while(len-- > 0)
   *p++ = *q++;
}
```

Common Implementations

The **restrict** qualifier specifies behavior that involves a relationship between one pointer and all other pointers. An alternative approach is to specify relationships between sets of pointers (which may only be a relationship between two pointers). Such an approach enables aliasing behaviors to be specified that would not be possible using the **restrict** qualifier. Koes, Budiu, Venkataramani and Goldstein[758] produced a tool which analyses source for sets of pointers which were likely, but not guaranteed, to be independent of each other and an estimated optimization benefit if a compiler could assume they were independent. They found that in many cases developers needed only a few minutes to verify whether the pointers were actually independent and that worthwhile increases in program speed were possible (they modified gcc to accept and use pointer information provided by a new pragma).

Coding Guidelines

Use of the **restrict** qualifier relies on the original use meeting, and subsequent developers maintaining, the guarantee required by the standard. While it may be possible to use static analysis tools to verify the requirement in some cases (where the analysis does not consume huge amounts of resources), the availability of such tools is open to question. Without a guaranteed method of enforcement, and no established practices for avoiding the problem, no guideline is given here.

Example

When defining a macro to take the place of a function, it is not necessary to be concerned with the scope of the argument passed, provided the lifetime of the pointer used is that of the body of the macro. In the following example, the claim being made by the restricted pointer only needs to exist within the block created by the macro body.

```
float *c;

#define WG14_N448(N, A, B)      \
```

```
    {                           \
    int n = (N);                \
    float *restrict a = (A);    \
    float *const b = (B);       \
    int i;                      \
    for (i = 0; i < n; i++)     \
       a[i] = b[i] + c[i];      \
    }
```

In the following two examples the objects accessed by the restricted pointers are subobjects of a larger whole. There can be so called *edge effects* where the two subobject storage areas meet. At this edge point it is
cache 0 possible that accesses to elements from both subobjects will be loaded into the same cache line. The affect may be that assumptions about cache line interaction, made when deciding what machine code to generate, are no longer true. The performance impact of optimizer assumptions about the cache not being met will be processor and optimizer specific.

```
void WG14_N866_D(int n, int * restrict x, int * restrict y)
{
/*
 * Let the number of elements in the pointed-to object be 3n.  The following
 * modifies the lower-n elements through x, and modifies the upper-n
 * elements through y.  The middle n elements are read through x and y.
 */
for(int i=0; i<n; i++)
   {
   x[i]     += x[i+n];
   y[i+2*n] += y[i+n];
   }
}

void f(void)
{
int z[300];

WG14_N866_D(100, z, z);
}
```

In the following example the individual array elements pointed to by each of the restricted pointers are disjoint.

```
void WG14_N886_K(int n, char * restrict p, char * restrict q)
{
for(int i=0; i<n; i+=2)
   {
   ++p[i];
   ++q[i];
   }
}

void all_edge_affects(void)
{
int r[100];

WG14_N886_K(100, r, r+1);
}
```

restrict intended use

The intended use of the **restrict** qualifier (like the **register** storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior). 1491

Commentary

The **restrict** type qualifier allows programs to be written so that translators can produce significantly faster executables. Anyone for whom this is not a concern can safely ignore this feature of the language. Rationale

A proof by Landi[804] brought the search for an optimal alias detection algorithm (determining whether two different identifiers refer to the same storage location at a particular point in a program) to an abrupt end (at least for programs containing **if** statements, loops, dynamic storage allocation, and manipulating at least a singly linked list). He showed that in general: (1) determining whether an alias occurs during some execution of a program is undecidable, and (2) determining whether an alias occurs during all executions of a program is uncomputable (simplifying the problem to (1) intraprocedural analysis where no use is made of dynamic storage allocation is NP-hard,[805] and (2) flow-insensitive, intraprocedural, may-alias analysis is NP-hard[596]). alias analysis

The **restrict** qualifier promotes optimization by reducing some costs (to vendors in producing the optimizer and the developer in terms of resources needed to run the translator), at the expense of increasing another cost (developer time needed to deduce where the qualifier can be added).

It is possible that deleting all instances of the **restrict** qualifier will change the observable behavior of a program (because the requirement on accesses was not being met). 1490 restrict requires all accesses

Other Languages

The implementations of some languages support pragmas that enable developers to communicate optimization information to the translator.

Common Implementations

Whenever a value is stored indirectly through a pointer, a translator has to assume that potentially all objects now hold a different values. A little analysis allows the set of potentially out-of-date object values to be reduced to those whose address is ever assigned to a pointer. Successively more sophisticated analyses can be used to reduce the size of this set. A store through a restricted pointer can be assumed to not affect the value of any other value accessed in the lifetime of that pointer object. This information helps other optimization techniques to do a better job, by making it possible for them to hold on longer to the information they have built up about the values and properties of objects.

The use of restricted pointers (rather than pointers without this qualification) is not itself an optimization technique. Use of a restricted pointer provides alias information that, for non-restricted pointer accesses, would otherwise have to be obtained through complete program analysis.

A number of algorithms for deducing the set of objects that a pointer could be pointing to, at any point in a program, have been proposed.[31, 870, 1510] Unfortunately, for all but the smallest programs, they require large amounts of memory and processor time. An empirical evaluation[577] of the different algorithms used programs of under 7.1 KLOC for its comparisons (this figure is an over estimate since it is based on a line count of the entire source, not just the executable statements), with a single program of 29.6 KLOC exceeding available memory (200 M byte) for some of the measurements. Researchers have started to make use of the common cases seen in production code to design alternative algorithms that require fewer resources. Making use of such information enabled an analysis of Microsoft Word 97,[318] 1.4 MLOC of C++, to produce results close to those obtained by Andersen's algorithm.[31]

Hind[576] discusses pointer analysis issues that still remain open.

Coding Guidelines

The **restrict** qualifier also has uses outside of optimization. The use of this qualifier could be treated as an assertion that could to be checked by static analysis tools. The C99 Standard is still too new to know if this kind of usage will occur.

Example

processor 940
SIMD

The SIMD approach uses a fine grained model of executing portions of a program in parallel. Being able to automatically break a program into components that execute on different processors (i.e., executing different functions on different processors) has proved to be extremely difficult. However, at the time of this writing there are no commercial translators offering such functionality. Automatically deducing that calls to `walk_tree`, in the following example, can be distributed to different processors remains a very difficult problem.

```
#include <stddef.h>

struct data {
             long important_value;
            };
struct t_rec {
              struct data information;
              struct t_rec * restrict left,
                           * restrict right;
             };
typedef struct t_ref * restrict TREE;

extern void process_data(struct data);

void walk_tree(TREE root)
{
if (root->left != NULL)
   walk_tree(root->left);
if (root->right != NULL)
   walk_tree(root->right);

process_data(root->information);
}
```

---

qualified array type

If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. 1492

Commentary

The implications of this specification become apparent when arrays are converted to pointers to their first element. The declaration `const int arr[10];` declares `arr` to have type array of **`const int`**. If an occurrence of this object is implicitly converted to a pointer type, the resulting type is pointer to **`const int`**. If the qualifier had been associated with the array type, the converted pointer type would have been **`const`** pointer to **`int`**. The following example requires a more complicated chain of reasoning (based on a discussion by McDonald):

```
struct T {
          int *mem_1[2];
          int **mem_2;
         };

const struct T s;  /* Members inherit const. */

void f(void)
{
TYPE_1 loc_1;
TYPE_2 loc_2;

loc_1 = &s.mem_1; /* Statement 1. */
```

```
14  loc_2 =  s.mem_1; /* Statement 2. */
15  }
```

What should the types TYPE_1 and TYPE_2 look like? The result of the dot selection operator includes any qualifiers in the type of its left operand. The two types needed are:

1033 struct qualified result qualified

1. an expression having array type is not converted to *pointer to . . .* when it is the operand of the unary **&** operator. Thus, `&s.mem_1` has type pointer to array of const pointer to int, and the declaration of `loc_1` needs to be `int * const (*loc_1)[2];`.

729 array converted to pointer

2. in the second case `s.mem_1` is implicitly converted to a pointer type. However, does the conversion to pointer type occur before the above C rule on qualifiers is applied? The two possibilities are:

729 array converted to pointer

```
1  int *  const *  /* Type if qualified before conversion. */
2  int ** const    /* Type if qualified after conversion.  */
```

   Applying the qualifier before conversion is considered to be the preferred interpretation (it is also the behavior of the implementations tested by your author; also see elsewhere) because it prevents the elements of the array, `mem_1`, being modified (whereas the member `mem_2` is treated as having type `int ** const`).

1497 EXAMPLE const aggregate

A method of specifying qualifiers, in an array declaration, so they are associated with the array type was introduced in C99 and is discussed elsewhere.

1571 qualified array of

Other Languages

Few, in any, languages implicitly convert arrays into pointers to their first element. So the distinction that occurs in C, because of this implicit conversion, does not arise.

Coding Guidelines

Because of the implicit conversion that occurs for parameters declared to have an array type, it is possible for them to be assigned to (because qualifiers applies to the element type). Assigning to an object defined, in the source, using an array type might be considered suspicious with or without a qualifier. However, the usage is rare and these coding guidelines are silent on the issue.

729 array converted to pointer

Example

```
1  void f(const int arr_1[10], const int arr_2[20])
2  {
3  arr_1 = arr_2;
4  }
```

1493 If the specification of a function type includes any type qualifiers, the behavior is undefined.[116)]

Commentary

A function type describes a function with specified return type. It would be meaningless to say that a function returned a volatile-qualified type. It only gets to return a single value. A function type having a const-qualified return type could be interpreted to mean that the value returned was always the same. The C committee choose not to give this interpretation to such a construct.

532 function type

C++

8.3.5p4

In fact, if at any time in the determination of a type a cv-qualified function type is formed, the program is ill-formed.

A C++ translator will issue a diagnostic for the appearance of a qualifier in a function type, while a C translator may silently ignore it.

The C++ Standard allows *cv-qualifiers* to appear in a function declarator. The syntax is:

8.3.5p1 *D1 (parameter-declaration-clause) cv-qualifier-seq$_{opt}$ exception-specification$_{opt}$*

the *cv-qualifier* occurs after what C would consider to be the function type.

Common Implementations

The most commonly seen behavior is to ignore the type qualifiers.

Coding Guidelines

Function types whose return type includes a qualifier can occur through the use of typedef names. It would be consistent for a functions return type to use the same typedef name as the objects appearing in its **return** statement. While it might be unusual for a const-qualified object to appear in a **return** statement, the usage appears harmless and is not common.

Example

footnote 137 1820 A footnote in the standard also provides some examples.

```
1  typedef const int C_I;
2
3  C_I f(int p_1)
4  {
5  return p_1 + 1;
6  }
```

---

qualified type to be compatible

For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; 1494

Commentary

simple assignment constraints 1296 The requirements on the types of the operands of the assignment operators ignore any qualifiers on the type of the right operand.

C++

compatible type if 631 The C++ Standard does not define the term *compatible type*. However, the C++ Standard does define the terms *layout-compatible* (3.9p11) and *reference-compatible* (8.5.3p4). However, *cv-qualifiers* are not included in the definition of these terms.

---

the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type. 1495

Commentary

type qualifier syntax 1476 Measurements of qualifier ordering are discussed elsewhere.

C++

The C++ Standard does not specify any ordering dependency on *cv-qualifiers* within a *decl-specifier*.

Coding Guidelines

declaration specifier ordering 1357.1 The guideline recommendation dealing with the ordering of keywords within type specifiers is discussed elsewhere.

1496 EXAMPLE 1 An object declared

const volatile EXAMPLE

```
extern const volatile int real_time_clock;
```

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

Commentary

Such an object may be in a memory mapped region of storage that is effectively read-only, from the programs perspective (in the sense that any writes to objects in that area of storage do not modify the values held there).

1497 EXAMPLE 2 The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

EXAMPLE const aggregate

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;      // valid
cs = ncs;      // violates modifiable lvalue constraint for =
pi = &ncs.mem; // valid
pi = &cs.mem;  // violates type constraints for =
pci = &cs.mem; // valid
pi = a[0];     // invalid: a[0] has type "const int *"
```

Commentary

The terms *valid* and *invalid* are commonly used by developers. However, the C Standard does not define them. As used in this example, the term *valid* might be interpreted as "does not affect the conformance status of the program". And the term *invalid* might be interpreted as "will cause a translator to issue a diagnostic message".

Coding Guidelines

Some developers might be surprised that in the following declarations:

```
1  const struct s { int mem; } cs = { 1 };
2  struct s ncs;  // the object ncs is modifiable
```

the **`const`** qualifier in the first declaration only applies to the type of `cs` (i.e., the tag `s` is not defined to refer to a const-qualified type). Type qualifiers are not part of any of the syntactic constructs used to define tags.

1464 struct tag; 1471 struct-or-union identifier not visible

1498 114) A **`volatile`** declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function.

footnote 114

Commentary

These are probably the two most common uses of the **`volatile`** qualifier.

C++

The C++ Standard does not make this observation.

1499 Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

Commentary

This is a requirement on the implementation. Accessing an object declared with a volatile-qualified type is a side effect. The result of this side effect is likely to be unknown to the implementation, which is likely to play safe.

185 side effect

C++

7.1.5.1p8 *[Note: **volatile** is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of **volatile** are intended to be the same in C++ as they are in C.]*

Example

In the function f below, a translator may optimize the evaluation of the expression appearing in the first return statement. The value 0 can be unconditionally returned after a read access to glob_2 (no multiplication operation is performed). The expression in the second **return** statement may exhibit undefined behavior. It depends on whether accessing glob_2 causes its value to be modified. An implementation could choose to unconditionally return the value 0 after two read accesses to glob_2 (no subtraction operation is performed). The undefined behavior, if an access does modify glob_2, being that the new values do not affect the value returned.

object 941 modified once between sequence points

```
1  extern int glob_1;
2  extern volatile int glob_2;
3
4  int f(void)
5  {
6  if (glob_1 == 1)
7     return glob_2 * 0;        /* First return.  */
8  else
9     return (glob_2 - glob_2); /* Second return. */
10 }
```

---

footnote 115

115) For example, a statement that assigns a value returned by **malloc** to a single pointer establishes this association between the allocated object and the pointer. 1500

Commentary

lvalue 725 converted to value

Taking the address of an object with static or automatic storage duration is not an access to that object. Such an address may also be assigned to a restricted pointer to establish this association.

---

footnote 116

116) Both of these can occur through the use of **typedef**s. 1501

Commentary

```
1  typedef char A[10];
2
3  volatile A v_a;
4
5  typedef int F(void);
6
7  const F *cp_f;
```

C++

The C++ Standard does not make this observation, but it does include the following example:

8.3.5p4 *[Example:*

```
typedef void F();
struct S {
const F f;     // ill-formed:
               // not equivalent to: void f() const;
};
```

—end example]

6.7.3.1 Formal definition of restrict

1502 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.

restrict formal definition

Commentary

A typedef name, function, and enumeration constant are also ordinary identifiers. However, they do not declare objects.

C++

Support for the **restrict** qualifier is new in C99 and is not available in C++.

Example

```
1  #include <stdlib.h>
2
3  typedef int T;
4
5  T * restrict D;
6
7  void f(void)
8  {
9  D = (T * restrict)malloc(sizeof(T)); /* Allocate object P. */
10 }
```

1503 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block.

restrict pointer lifetime

Commentary

This and the following two sentences define a region of program text, denoted by **B** (and **B2** elsewhere). This formal definition of **restrict** refers to events that occur before, during, or after the execution of this block.

1512 restrict B2
1514 restrict execution of block means

Example

```
1  typedef int T;
2
3  void f(void)
4  {               /* Started D's associated block B.  */
5  T * restrict D  /* D is now visible in the block B. */
6                  ;
7  }               /* Ended D's associated block B.    */
```

1504 If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block.

Commentary

```
typedef int T;

void f(T * restrict D /* D is now visible, outside of its associated block B. */
                      )
{                      /* Started D's associated block B. */
/* ... */
}                      /* Ended D's associated block B.   */
```

Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment). 1505

Commentary

The cases covered by this *otherwise* include all objects having static storage duration and file scope, and all objects having allocated storage duration.

---

restrict pointer based on

In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.[117] 1506

Commentary

additive operators 1165 pointer to object

This defines the term *based*. The use of the term *array object* here refers to a prior specification about a pointer to a non-array object behaving like a pointer to an array containing a single element (although technically that wording limits itself to "for the purposes of these operators").

That is, the value of **E** depends on the value of **P**, not the value pointed to by **P**. It is possible that the former value of **P** is indeterminate, so the involvement of copies of formerly pointed-to array-objects is a conceptual one.

Example

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

struct t {
          int *q;
          int i;
         } a[2] = { 0, /* ... */ };

void WG14_N448(struct t *restrict P,
               _Bool c_flag)
{                                /* Block B, associated with P, has started here. */
struct t *q;
intptr_t n;
/*
 * Adjust P to point at copy of original object?
 */
if (c_flag)
   {
   struct t *r;
   r = malloc(2*sizeof(*P));
   memcpy(r, P, 2*sizeof(*P));
   P = r;
   }
```

```
q = P;
n = (intptr_t)P;
/*
 * Lists of pointer expressions that are, and are not,
 * based on object P, in the execution of block B.
 *
 *      expression E           expression E
 *    based on object P     not based on object P
 *
 *        P                     P->q
 *        P+1                   P[1].q
 *        &P[1]                 &P
 *        &P[1].i
 *        &P->q
 *        q                     q->q
 *        ++q
 *        (char *)P             (char *)(P->i)
 *        (struct t *)n         ((struct t *)n)->q
 */
}                                /* Block B has ended here. */

void f(void)
{
WG14_N448(a, 0);
WG14_N448(a, 1);
}
```

1507 Note that "based" is defined only for expressions with pointer types.

Commentary

For instance, in the example function, `WG14_N448`, given in the Example subclause of the previous C sentence, *based* is defined for `q->q`, but not for `q->i`.

1508 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**.

restrict
&L based on L

Commentary

This defines the term **L**. Starting with the address of **L**, that is based on **P**, we can use it to walk back through any chain of pointers.

Example

In the following we have `&(p->v[i])` (equivalent to `(p->v)+i`) based on the restricted pointer `p->v`. Walking up the pointer chain, `&(p->v)` is based on the restricted pointer `p`,

```
typedef struct {
                 int n;
                 double * restrict v;
                } vector;

void f(vector * restrict p)
{
p->v[i];
}
```

1509 If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified.

Commentary

This requirement means that, if **T** is const-qualified then this qualification cannot be cast away and the object **X** modified via some non-const-qualified lvalue **L** (if this sequence of operations occurred the behavior would be undefined).

Example

The use of **const** (via the typedef name T) and **restrict** in the definition of WG14_N866_F implies that g only accesses the value of *p, and does not modify it. On some processor architectures a translator could use this information to generate code to initiate the load from *p before the call to g, making the value available sooner both within execution of g and for the **return** statement. The size of the performance benefit will depend on the size of the type T.

```
1  typedef const int T; /* Some type T. */
2  extern void g();
3
4  T WG14_N866_F(T * restrict p)
5  {
6  g(p);
7  return *p;
8  }
```

Every other lvalue used to access the value of **X** shall also have its address based on **P**. 1510

Commentary

Automatically deducing, during program translation, the set of lvalues used to access X is potentially very expensive, computationally. Being able to assume that such lvalues are based on P (otherwise the behavior is undefined) considerably reduces an optimizer's computational overhead (when performing optimizations that rely on points-to information).

Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. 1511

Commentary

There are a number of ways wording in the standard can promote optimization. Possibilities include: (1) bounding the behavior for common cases (such as limits on how objects referenced by restricted pointers are accessed), and (2) ensuring that edge cases have undefined behavior (a translator does not have to bother taking them into account, the behavior can be whatever happens to occur). The purpose of this C sentence is to render some awkward edge cases undefined behavior.

Example

When the one or more parameters of a function definition have a restricted pointer type a translator needs to be able to optimize the function body, independent of the actual arguments passed. Passing the address of an object as the value of two separate arguments, in a function call, occurs reasonably frequently. The following examples are based on a discussion in document WG14/N866 written by Homer.

restrict 1519 example 2

```
1  typedef struct {
2                  int n;
3                  double * restrict v;
4                 } vector;
5
6  void WG14_N866_C(vector * restrict p, vector * restrict q)
7  {
8  int i;
9  for(i=1; i < p->n; i++)
10     p->v[i] += q->v[i-1];
11 }
```

What behavior does this C clause give to the calls `WG14_N866_C(&x, &y)` and `WG14_N866_C(&x, &x)` (when `x.n>1`)?

1. **B** is the block formed by the body of the definition of `WG14_N866_C`,
2. Let **L** be the lvalue `p->v[i]`, and **X** the object which it designates,
3. Let **X** be `p->v[i]`,
4. `&(p->v[i])` (equivalent to `(p->v)+i`) is based on the restricted pointer `p->v`, 1508 restrict &L based on L
5. a modification of **X** is considered also to modify the object designated by `p->v` (current C sentence),
6. recursively, `&(p->v)` is based on the restricted pointer `p`,
7. when the arguments passed are `(&x, &y)` `p` and `q` refer to different objects, and no other pointer is based on `p`,
8. when the arguments are `(&x, &x)` the same object is also accessed through the lvalue `q->v`, but `&(q->v)` is not based on `p` and therefore behavior is undefined. 1513 restrict undefined behavior

A translator is thus able to unconditionally optimize the body of the function `WG14_N866_C` (either the **`restrict`** semantics does hold, or the undefined behavior can be that it does :-).

In the following function definition, even if `*a` and `*b` happen to refer to the same object, the members `a->p` and `b->q` will be distinct restricted pointers. A translator can infer that `*(a->p)` and `*(b->q)` are distinct objects, and so there is no potential aliasing to inhibit optimization of the body of `f`.

```
1  typedef struct {
2                   int * restrict p;
3                   int * restrict q;
4                  } T;
5
6  int WG14_N866_E(T * restrict a, T * restrict b)
7  {
8  *(a->p) += 1;
9  return *(b->q);
10 }
```

During an execution of `f`, the only object that is actually modified is `(*a)->p`. Now `&((*a)->p)` is `a->p`, a restricted pointer, so `a->p` is also considered to be modified. Recursively, `&(a->p)` is based on `a`, another restricted pointer. Because no other lvalues are used to access either `(*a)->p` or `a->p`, all the requirements of the specification are met (and they do not involve `b`).

1512 If **`P`** is assigned the value of a pointer expression **`E`** that is based on another restricted pointer object **`P2`**, associated with block **`B2`**, then either the execution of **`B2`** shall begin before the execution of **`B`**, or the execution of **`B2`** shall end prior to the assignment. restrict B2

Commentary

This requirement allows restricted pointers to be passed as arguments to function calls, and a restricted pointer within a nested scope to be assigned the value of a restricted pointer from an outer scope. If the value of a restricted pointer is assigned to another restricted pointer, e.g., **P_nest**, in a nested scope, the requirement applies only within the scope of **P_nest** and only to objects modified within that scope that are referenced through expressions based on **P_nest** at least once.

The case where execution of **B2** ends prior to the assignment covers the situation where a value, based on the restricted pointer object **P2**, returned from a function call is assigned to **P**. 1521 restrict example 4

Coding Guidelines

Parallels can be drawn between the coding guideline discussion on the assignment of object addresses to pointers and the assignment of restricted pointers to other restricted pointer objects. 454 pointer indeterminate

Example

```
#define INLINE(P_1, P_2, N) {                          \
                            int * restrict q_1 = P_1;  \
                            int * restrict q_2 = P_2;  \
                            for (int i = 0; i < N; i++)  \
                               {                       \
                               q_1[i] += q_2[i];       \
                               }                       \
                            }
```

---

restrict undefined behavior

If these requirements are not met, then the behavior is undefined. 1513

Commentary

In particular, an implementation is free to make the general assumption that the requirements are met and to perform the optimizations that are performed when they are met.

---

restrict execution of block means

Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with **B**. 1514

Commentary

lifetime 451 of object

This sentence clarifies the meaning by using an existing, well defined, term, lifetime.

A translator is free to ignore any or all aliasing implications of uses of **restrict**. 1515

Commentary

A developer may be able to deduce translator behavior from the performance of the program image, or by examining the undefined behavior that occurs for certain constructs.

restrict 1513 undefined behavior

Coding Guidelines

Uses of **restrict** provide information on source code properties believed to be true by developers. Something that static analysis tools can check and make use of.[10]

EXAMPLE 1 The file scope declarations 1516

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of **a**, **b**, or **c**, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

Commentary

The declaration of c does not explicitly assert anything about **restrict** related semantics. It just so happens that all of the other declarations are restrict-qualified, which implies additional semantics on accesses to c. If another declaration, say `extern int *d;`, appeared in the list, then the same object could be accessed using either c or d, but the stated relationship between a, b, and c would remain true (and so would an equivalent one between a, b, and d).

---

footnote 117

117) In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced indirectly through **P**. 1517

Commentary

The specification of **restrict** provides a mechanism for developers to give guarantees about the set of objects a pointer may point at. Translators may use these guarantees to deduce information about the value of an object referenced indirectly though so qualified pointers.

1518 For example, if identifier **p** has type **(int **restrict)**, then the pointer expressions **p** and **p+1** are based on the restricted pointer object designated by **p**, but the pointer expressions ***p** and **p[1]** are not.

Commentary

If p had type (int * restrict * restrict then both *p and p[1] would be based on the restricted pointer *p.

1519 EXAMPLE 2 The function parameter declarations in the following example

restrict example 2

```
void f(int n, int * restrict p, int * restrict q)
{
        while (n-- > 0)
                *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other.

The benefit of the **restrict** qualifiers is that they enable a translator to make an effective dependence analysis of function **f** without examining any of the calls of **f** in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of **f** in **g** has undefined behavior because each of **d[1]** through **d[49]** is accessed through both **p** and **q**.

```
void g(void)
{
        extern int d[100];
        f(50, d + 50, d); // valid
        f(50, d +  1, d); // undefined behavior
}
```

Commentary

As this example states, the burden of correctness lies with the programmer. A translator is entitled to act as if what the programmer said (through the use of **restrict**) is correct.

Other Languages

Some languages (e.g., Fortran and Ada) copying operations such as these can be performed through the use of array slicing operators.

Coding Guidelines

The technical difficulties involved in proving that a developer's use of **restrict** has defined behavior are discussed elsewhere.

1491 alias analysis

1520 EXAMPLE 3 The function parameter declarations

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
        int i;
        for (i = 0; i < n; i++)
                p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if **a** and **b** are disjoint arrays, a call of the form **h(100, a, b, b)** has defined behavior, because array **b** is not modified within function **h**.

Commentary

Information on when objects may, or may not, be modified is of primary importance to the optimization process. Lack of information on when two read accesses refer to the same object may result in suboptimal code, but it does not prevent some optimizations being made. The committee did not consider it worth addressing this "reads via two apparently independent pointers" issue in C99.

restrict example 4

EXAMPLE 4 The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only "outer-to-inner" assignments between restricted pointers declared in nested blocks have defined behavior. 1521

```
{
        int * restrict p1;
        int * restrict q1;
        p1 = q1; // undefined behavior
        {
                int * restrict p2 = p1; // valid
                int * restrict q2 = q1; // valid
                p1 = q2;                // undefined behavior
                p2 = q2;                // undefined behavior
        }
}
```

The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits **new_vector** to return a **vector**.

```
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
        vector t;
        t.n = n;
        t.v = malloc(n * sizeof (float));
        return t;
}
```

Commentary

restrict B2 1512 These exceptions are discussed elsewhere.

6.7.4 Function specifiers

function specifier syntax

1522

function-specifier:
inline

Commentary

Rationale The **inline** keyword, adapted from C++ . . .

The keyword **inline** is invariably used by languages to specify functions that are to be considered for inlining by a translator.

C90

Support for *function-specifier* is new in C99.

C++

The C++ Standard also includes, 7.1.2p1, the *function-specifier*s **virtual** and **explicit**.

Other Languages

Few languages specify support for function inlining, although their implementations may provide it as an extension (e.g., some Fortran implementations support some form of `inline` command line option). Some implementations support automatic inlining as an optimization phrase. CHILL supports the **`inline`** keyword. Ada defines a pragma directive that may be used to pass inlining information to the translator, e.g., `pragma inline(fahr)`.

Common Implementations

`gcc` supported **`inline`** as an extension to C90.

Constraints

1523 Function specifiers shall be used only in the declaration of an identifier for a function.

Commentary

The concept denoted by the only available function specifier has no interpretation for object or incomplete types.

1529 inline suggests fast calls

Coding Guidelines

Those coding guideline documents that argue against the use of the **`register`** storage-class specifier may well argue against the use of function specifiers for the same reasons. These coding guidelines do not recommend against this usage for the same reason they did not recommend against the use of the **`register`** storage-class specifier.

1370 register extent effective

1524 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage.

inline static storage duration

Commentary

Note: this constraint applies to an *inline definition*. Having a function declared using the **`inline`** specifier is not enough for this constraint to apply.

1540 inline definition

1541 inline definition not an external definition

Functions declared with external linkage and the inline specifier can have more than one definition. Two function definitions containing any of the identifiers described by this constraint could not be called interchangeably. For instance, given the translation unit:

file_1.c

```
1  #include "f.h"
2
3  void g_1(void)
4  {
5  f();
6  }
```

and another translation unit:

file_2.c

```
1  #include "f.h"
2
3  void g_2(void)
4  {
5  f();
6  }
```

and the header file:

f.h

```
1  inline void f(void)
2  {
```

```
3  static int total;
4
5  total++;
6  }
```

the call to the function `f` within `file_1.c` could result in the inline function defined in that source file being invoked, while the call in `file_2.c` could result in the inline function defined in that source file being invoked. C has a very loose model for handling translation of separate source files, a translator is not required to have knowledge of the contents of `file_1.c` when it is translating `file_2.c` and vice versa. In the above case storage for two copies of `total` will be created and independently referenced. This is likely to be surprising to developers who carefully ensure that only one definition of `f` exists.

linkage 420

The requirements in this constraint prevent developers being surprised by this case and do not require any separate source file translation techniques beyond those currently required of a translator.

An object defined to be a const-qualified type is not modifiable and hence the following is permitted:

modifiable 724
lvalue

f.h

```
1  inline void f(void)
2  {
3  static const int magic = 42;
4  }
```

Rationale

Therefore, the following example might not behave as expected.

```
inline const char *saddr(void)
{
    static const char name[] = "saddr";
    return name;
}

int compare_name(void)
{
    return saddr() == saddr();  // unspecified behavior
}
```

Since the implementation might use the inline definition for one of the calls to `saddr` and use the external definition for the other, the equality operation is not guaranteed to evaluate to 1 (true). This shows that static objects defined within the inline definition are distinct from their corresponding object in the external definition. This motivated the constraint against even defining a non-**`const`** object of this type.

C++

The C++ Standard does not contain an equivalent prohibition.

7.1.2p4 *A **`static`** local variable in an **`extern inline`** function always refers to the same object.*

The C++ Standard does not specify any requirements involving a static local variable in a static inline function. Source developed using a C++ translator may contain inline function definitions that would cause a constraint violation if processed by a C translator.

Coding Guidelines

The guideline recommendation dealing with identifiers at file scope that are referenced in more than one source file is applicable here.

identifier declared in one file 422.1

In a hosted environment, the **`inline`** function specifier shall not appear in a declaration of **`main`**. 1525

Commentary

An implementation may associate special properties with the function called on program startup. In a hosted environment the name of the function called at program startup is known (i.e., `main`). While in a freestanding environment the name of this function is not known (it is implementation-defined).

162 hosted environment startup
155 freestanding environment startup

C++

A program that declares `main` *to be* **`inline`** *or* **`static`** *is ill-formed.* 3.6.1p3

A program, in a freestanding environment, which includes a declaration of the function `main` (which need not exist in such an environment) using the **`inline`** function specifier will result in a diagnostic being issued by a C++ translator.

Semantics

1526 A function declared with an **`inline`** function specifier is an *inline function*. inline function

Commentary

This defines the term *inline function*, which is a commonly used term, by developers, in many languages.

1527 The function specifier may appear more than once; function specifier appears more than once

Commentary

This permission is consistent with that given for type qualifiers.

1479 qualifier appears more than once

C++

The C++ Standard does not explicitly specify this case (which is supported by its syntax).

Coding Guidelines

The coding guideline issues for function specifiers occurring more than once are different from those for type qualifiers. There is a single function specifier and the context in which it occurs creates few rationales for the need of more than one to appear. However, while multiple function specifiers might be considered unusual there does not appear to be any significant benefit in a guideline recommendation against this usage.

1479 qualifier appears more than once

1528 the behavior is the same as if it appeared only once.

Commentary

This is consistent with type qualifiers.

1529 Making a function an inline function suggests that calls to the function be as fast as possible.[118)] inline suggests fast calls

Commentary

The **`inline`** specifier is a hint from the developer to the translator. The 1970s gave us the **`register`** keyword and the 1990s have given us the **`inline`** keyword. In both eras commercially usable translation technology was not up to automating the necessary functionality and assistance from developers was the solution adopted. Published figures on translators that automatically decide which functions to inline[216, 323, 348] show total program execution time reductions of around 2% to 8%. The latest research on inlining[1449] has not significantly improved on these program execution time reductions, but it does not seem to cause the large increase in executable program size seen in earlier work, and the translation overhead associated with deducing what to inline has been reduced.

The complexity of selecting which functions to inline, to minimize a program's execution time while keeping the size of the program image below some maximum limit and the size of individual functions below some maximum limit, is known to be at least NP-complete.[1199]

141 program image

Developers often believe that function calls take a long time to execute (relative to other instructions). While this might have been true 20 years ago, it is rarely true today. Processor designers have invested significant resources in speeding up the calling of functions (call instructions have been simplified, no complicated VAX type instructions,[284] and branch prediction is applied to the call/return instructions, helping to minimize stalls of the instruction pipeline). See Davidson[323] for a performance comparison of the effect of **inline** across four different processors.

processor 0 pipeline

On modern processors the time taken to execute a call or return instruction is usually less than that required to execute a multiply (although it may slow the execution of other instructions because of a stalled pipeline or failed branch prediction), it can even be as fast as an add instruction. The relative unimportance of call/return instruction performance is shown both by situations where a dramatic decrease in execution time has little impact on overall program performance[1435] and the fact that putting duplicate sections of code in a translator created function is treated as a potentially worthwhile speed optimization.[786] The impact of using separate functions, rather than inlined code, is in areas other than the call/return instruction execution overhead. They include the instruction cache, register usage, and the characteristics of memory (locals on the stack) accesses.

branch prediction 1739

It often comes as a surprise to developers that use of the **register** storage class can slow a program down. The same is also true of the **inline** function specifier; its use can slow a program down (although the situations where this occurs appear to be less frequent than for the **register** storage class). Degradations in performance due to an increase in page swapping (on hosts with limited storage) or an increase in program size causing a decrease in the number of cache hits are the most commonly seen reasons. One published report[275] (Fortran source) found that a lack of sophisticated dependency analysis in the translator meant that it had to make worst-case assumptions in a critical loop that did not apply in the non-inlined source. Even when **inline** is used intelligently (based on execution counts of function calls) improvements in performance can vary significantly, depending on the characteristics of the source code and on the architecture of the host processor.[323, 348]

register 1369 storage-class

C++

The C++ Standard gives an implementation technique, not a suggestion of intent:

7.1.2p2 *The **inline** specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism.*

Such wording does not prevent C++ implementors interpreting the function specifier in the C Standard sense (by, for instance, giving instructions to the hardware memory manager to preferentially keep a function's translated machine code in cache).

Common Implementations

A parallel can be drawn with the hint provided by the **register** storage class. To what extent will an implementation unconditionally follow the suggestion provided by the appearance of the **inline** function specifier on a function definition, ignore it completely (performing its own analysis of program behavior to deduce which function calls should be inlined[216]), or implement some half-way point? Support for this keyword is new in C99 and there are still too few implementations supporting it to be able to spot any trends.

register 1369 storage-class

Although inlining is thought of in terms of speeding up the call itself, removing the machine instruction that performs the call is often the smallest saving made. Other savings are obtained from the removal of the interfacing machine code that saves and restores registers across the call, and the code for creating a new stack frame and restoring the old one on return (Davidson[323] gives equations for making various cost/benefit decisions and compares predicted behavior against results obtained from four different processors).

Inlining can also have effects beyond the immediate point of call. Many translators treat a single function as the unit of optimization, making worst-case assumptions about the effects of any function calls. Inlining a function allows the statements it contains to be optimized in the context of the call site (arguments are often constant expressions and their values can often replace parameter object accesses in the function body) and also allows information that previously had to be thrown away, because of the call, to be kept and used (a

recent innovation, known as *cloning*, changes calls to a function with a call to a copy of that function that has been optimized, based on knowledge of the arguments passed;[67, 1449] when two or more calls to a function share some argument values this technique can provide almost the same performance improvement without the overhead of excessive code expansion).

Inlining a function at the point of call can have disadvantages (and potentially no advantages), including the following:

- The quantity of generated code can increase significantly. Storage to hold generated code is rarely a problem on hosted implementation, but in freestanding implementations it can be a major issue. The increase in size of a program image can also affect the performance of processors instruction cache; the possible effects are complex, depending on size and configuration of the cache.[221, 916] The Texas Instruments TMS320C compiler[623] supports the `-io size` option. The optimizer multiplies the number of times a function is called by its size (an internal, somewhat arbitrary, measure is used) and only inlines the function if the result is less than the developer specified value of `size`.
- The maximum amount of stack storage required by a program can increase. When a function is inlined its stack storage requirements are added to those of the function into which it is merged. Optimizers choose not to inline functions at some call sites if the increase in stack storage requirements exceeds some predefined limit.[216]

```
inline void f(void)
{
int af[100];
/* ... */
}

void g(void)
{
int ag[100];
/* ... */
}

void h(void)
{
/*
 * When f is not inlined the storage it uses is freed before
 * the call to g, and so can be reused.
 * When f is inlined into h the storage it uses becomes part of
 * the storage allocated to h, and additional storage is required
 * by the call to g.
 */
f();
g();
}
```

Inlining not only offers opportunities for reducing the mount of generated code, but also reducing the total amount of stack storage required by nested function calls. Objects with automatic storage duration need to be allocated storage whether the function that defines them is inlined or not. When the function is inlined the storage is allocated in the stack frame of the function into which they are inlined. Inlining thus changes the stack usage profile of a program. Storage requirements can either increase or decrease, depending on what functions are inlined, the housekeeping overhead of a function call and the extent to which it is possible for objects to share storage locations.

Ratliff[1146] modified `vpcc`[326] to attempt to minimize the amount of stack frame storage required for locally defined objects (by using the same storage for different objects, based on the regions of code over which those objects were accessed; the SPARC architecture was used, which has alignment requirements). Table 1529.1 shows the affect of inlining on the amount of storage that is saved.

Table 1529.1: Number of bytes of stack space needed by various programs before and after inlining (automatically performed by vpcc). *Bytes saved* refers to the amount of storage saved by optimizing the allocation of locally defined objects. Adapted from Ratliff.[1146]

Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)	Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)
ackerman	312	8 (2.56)	232	8 (3.45)	linpack	1,504	48 (3.19)	3,312	112 (3.38)
bubblesort	568	8 (1.41)	136	8 (5.88)	mincost	1,216	0 (—)	192	8 (4.17)
cal	384	0 (—)	96	0 (—)	prof	1,584	0 (—)	400	40 (10.00)
cmp	768	0 (—)	192	0 (—)	sdiff	2,536	0 (—)	5,784	16 (0.28)
csplit	1,488	0 (—)	728	0 (—)	spline	560	8 (1.43)	200	8 (4.00)
ctags	8,144	0 (—)	24,544	88 (0.36)	tr	192	0 (—)	96	0 (—)
dhrystone	664	0 (—)	200	8 (4.00)	tsp	3,008	8 (0.27)	2,216	56 (2.53)
grep	592	0 (—)	304	0 (—)	whetstone	568	0 (—)	488	296 (60.66)
join	480	0 (—)	96	0 (—)	yacc	4,232	0 (—)	1,360	8 (0.59)
lex	9,472	0 (—)	7,208	8 (0.11)	average	1,989	4 (0.47)	2,510	34 (5.23)

Automatically inlining all functions can lead to very large program images. While heuristics based on number of calls and function size can reduce code expansion, information on which functions are frequently called during program execution enables a more targeted approach to inlining to be made (see Arnold, Fink, Sarkar, and Sweeney[59] for a comparison of inlining performance based on using static and dynamic information, Java based).

The translator for the HP–was DEC– Tru68 platform supports the **__forceinline** storage-class modifier.[600] Functions declared using this modifier are unconditionally inlined by the translator.

Coding Guidelines

The term *type safe macro* (because the types of the arguments are checked) or simply *safe macro* (because the arguments are only evaluated once) are sometimes applied to the use of inline functions.

The extent to which such suggestions are effective is implementation-defined.[119)] 1530

Commentary

There is no requirement on an implementation to handle calls to a function defined with the **inline** function specifier any differently than calls to a function defined without one. This behavior parallels that for the **register** storage-class specifier.

register 1370 extent effective

C++

7.1.2p2 *An implementation is not required to perform this inline substitution at the point of call;*

A C++ implementation is not required to document its behavior.

Coding Guidelines

register 1370 extent effective

Drawing parallels with the implementation-defined behavior of the **register** storage class would suggest that although this behavior is implementation-defined, no recommendation against its use be given. However, there is an argument for recommending the use of **inline** in some circumstances. Developers sometimes use macros because of a perceived performance advantage. Suggesting that an inline function be used instead may satisfy the perceived need for performance (whether or not the translator used performs any inlining is often not relevant), gaining the actual benefit of argument type checking and a nested scope for any object definitions.

Any function with internal linkage can be an inline function. 1531

Commentary

The C Standard explicitly gives this permission because it goes on to list restrictions on inline functions with external linkage.

C++

The C++ Standard does not explicitly give this permission (any function declaration can include the **inline** specifier, but this need not have any effect).

1532 For a function with external linkage, the following restrictions apply:

Commentary

The following restrictions define a model that has differences from the one used by C++.

Rationale

Inlining was added to the Standard in such a way that it can be implemented with existing linker technology, and a subset of C99 inlining is compatible with C++.

C++

The C++ Standard also has restrictions on inline functions having external linkage. But it does not list them in one paragraph.

A program built from the following source files is conforming C, but is ill-formed C++ (3.2p5).

File a.c

```
inline int f(void)
{
return 0+0;
}
```

File b.c

```
int f(void)
{
return 0;
}
```

Building a program from sources files that have been translated by different C translators requires that various external interface issues, at the object code level, be compatible. The situation is more complicated when the translated output comes from both a C and a C++ translator. The following is an example of a technique that might be used to handle some inline functions (calling functions across source files translated using C and C++ translators is more complex).

x.h

```
inline int my_abs(int p)
{
return (p < 0) ? -p : p;
}
```

x.c

```
#include "x.h"

extern inline int my_abs(int);
```

The handling of the second declaration of the function `my_abs` in `x.c` differs between C and C++. In C the presence of the **extern** storage-class specifier causes the definition to serve as a non-inline definition. While in C++ the presence of this storage-class specifier is redundant. The final result is to satisfy the requirement for exactly one non-inline definition in C, and to satisfy C++'s one definition rule.

1350 C++ one definition rule

If a function is declared with an **inline** function specifier, then it shall also be defined in the same translation unit. 1533

Commentary

This is a requirement on the program. It removes the need for linker technology that obtains a definition from some place other than the same translation unit.

C++

7.1.2p4 *An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).*

The C++ Standard only requires the definition to be given if the function is used. A declaration of an inline function with no associated use does not require a definition. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

Coding Guidelines

identifier declared in one file 422.1

The guideline recommendation dealing with placing the textual declaration of identifiers, having external linkage, in a single source file is applicable here.

footnote 118

118) By using, for example, an alternative to the usual function call mechanism, such as "inline substitution". 1534

Commentary

Another possibility would be to load the machine code generated from a function body into faster memory, for instance cache memory.

Other Languages

Most language specifications do not discuss inline function implementation details.

Common Implementations

Most translators perform inlining using some internal representation (which is rarely viewable). The output of a function inliner implemented by Davidson and Holler[322] was C source code (with functions having been inlined at the point of call).

Inline substitution is not textual substitution, nor does it create a new function. 1535

Commentary

A number of interpretations of the term *inline substitution* are possible. This wording clarifies that the two listed interpretations (with their associated semantics) are not intended to apply. From the developers point of view the semantic of a function call are the abstract one specified by the standard. The only measurable external changes, caused by the addition of a *function-specifier* to the definition of a function, in a program image should be in its size and execution time performance.

operator () 1000

An inline function is processed through the eight phases of translation at their point of definition. The sequence of machine code instructions used to implement a call to that function is an internal detail that is not visible to the developer. A few changes have to be made to the machine code when it is inlined. A **return** statement that contains a value has to be changed so that the value is simply treated like any other expression that occurred at the point of the function call. Any invocation of the `va_*` macros still have to refer to the arguments of the function that originally contained them (which may mean creating a dummy function call stack).

C++

The C++ Standard does not make this observation.

Coding Guidelines

Inline functions are too new to know whether developers make either of these incorrect assumptions.

1536 Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called;

Commentary

Any other specification of behavior has the potential to cause a change in the appearance of the **inline** function specifier, in a function definition, to change the semantics of a program. Also the existing translation model uses phases of translation. Macro expansion is performed in translation phase 4, prior to any identifiers being converted to keywords.

129 translation phase 4

C++

The C++ Standard does not make this observation.

1537 and identifiers refer to the declarations in scope where the body occurs.

Commentary

The identifiers that may be in scope where the body occurs, but not where the call to the function occurs all have file scope. They include typedef names and tag names, as well as objects and other functions.

1538 Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

Commentary

It is a requirement on the implementation that the address of every function, during program execution, be unique (also see the response to DR #079). It is also a requirement on the implementation that pointers to different functions do not compare equal.

422 function external linkage denotes same
423 identifier same if internal linkage
1233 pointers compare equal

C++

*An **inline** function with external linkage shall have the same address in all translation units.* 7.1.2p4

There is no equivalent statement, in the C++ Standard, for inline functions having internal linkage.

1539 119) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **inline** declaration.

footnote 119

Commentary

A translator that operates in a single pass over the source, which the majority do, does not have access to the body of an inline function until its definition is encountered. Consequently it may decide that any calls prior to the definition are not inlined. Other complications that might cause a translator to not perform inline substitution include the following:

10 implementation single pass

- When one or more functions forms a recursive chain it may be difficult to fully inline one function into any of the others.
- The `va_*` macros make assumptions about how the parameters they refer to are laid out in storage. The overhead of ensuring that these layout requirements are met, in any inline function that contains uses of these macros, may be sufficient to prevent inline substitution providing any benefit.

inline definition

If all of the file scope declarations for a function in a translation unit include the **inline** function specifier without **extern**, then the definition in that translation unit is an *inline definition*. 1540

Commentary

EXAMPLE 1544
inline
implementation 10
single pass
as-if rule 122
inline 1524
static storage duration

This defines the term *inline definition*. This specification violates the general principle that it be possible to translate C in a single pass. The presence of the word *all* means that a translator has to have seen all declarations of an identifier before it knows whether it is an inline definition. While an implementation still may choose to perform inline substitution before it has processed all of the source (subject to the as-if rule), some constraint requirements only apply to inline definitions, not external definitions, (an implementation is not prohibited from generating spurious diagnostic messages, but it must still correctly translate the source file).

C++

This term, or an equivalent one, is not defined in the C++ Standard.

The C++ Standard supports the appearance of more than one inline function definition, in a program, having a declaration with **extern**. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

Coding Guidelines

identifier 422.1
declared in one file

If the guideline recommendation specifying a single textual definition of an identifier is followed there will never be more than one declaration for a function, in a translation unit, that include the **inline** function specifier.

Example

```
1  inline int f_1(void)         /* May be an inline definition. */
2  {}
3  inline int f_2(void)         /* May be an inline definition. */
4  {}
5  extern inline int f_3(void)  /* Not an inline definition. */
6  {}
7  extern inline int f_1(void); /* No, f_1 is not an inline definition. */
8
9                               /* End-of-File, f_2 is an inline definition. */
```

inline definition not an external definition

An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. 1541

Commentary

external 1817
definition

An inline definition is explicitly specified as not being an external definition. The status of an inline function as an inline definition or an external definition does not affect the suggest it provides to a translator.

external 1818
linkage
exactly one external definition

An inline definition is intended for use only within the translation unit in which it occurs. Because it is not an external definition, the constraint requirement that only one external definition for an identifier occur in a program does not apply. However, if a function is used within an expression an external definition for it must exist somewhere in the entire program. The absence, in a function declaration, of **inline** or the inclusion of **extern** in a declaration creates such an external definition. Also, because an inline definition does not provide an external definition, another translation unit may have to contain an external definition (to satisfy references from other translation units).

An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. 1542

Commentary

The inline definition provides all the information (return type and parameter information) needed to call the external definition. Its body can also be used to perform inline substitution. An inline function might be said to have *ghostly linkage*. It exists if the translator believes in it. Otherwise it does not exist and the external definition is referenced.

C++

In C++ there are no alternatives, all inline functions are required to be the same.

> *If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.* 7.1.2p4

A C program may contain a function, with external linkage, that is declared inline in one translation unit but not be declared inline in another translation unit. When such a program is translated using a C++ translator a diagnostic may be issued.

Coding Guidelines

The guideline recommendation specifying a single textual definition is applicable here. 422.1 identifier declared in one file

---

1543 It is unspecified whether a call to the function uses the inline definition or the external definition.[120)]

Commentary

The C Standard does not require that the sequence of tokens representing an inline definition or external definition, of the same function, be the same. However, the intended implication, to be drawn from the unspecified nature of the choice of the definition used, is that a programs external output shall be the same in both cases (apart from execution time performance). If the definitions of the two functions are such that the external program outputs would not be the same, the behavior is undefined. 49 unspecified behavior

C++

In the C++ Standard there are no alternatives. An inline definition is always available and has the same definition:

> *An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).* 7.1.2p4

Rationale

> Second, the requirement that all definitions of an inline function be "exactly the same" is replaced by the requirement that the behavior of the program should not depend on whether a call is implemented with a visible inline definition, or the external definition, of a function. This allows an inline definition to be specialized for its use within a particular translation unit. For example, the external definition of a library function might include some argument validation that is not needed for calls made from other functions in the same library. These extensions do offer some advantages; and programmers who are concerned about compatibility can simply abide by the stricter C++ rules.

Common Implementations

The context in which a call occurs can have as much influence over whether a function is inlined as the definition of the function itself. For instance, are there any non-inlined calls nearby (which would have already prevented any flow analysis from building up much information that could be used at the inlined call), does the size of the function exceed some specified, internal translator, limit (there can be other limits such as the amount of storage required by the declarations in a function)?

Coding Guidelines

identifier declared in one file 422.1

If the guideline recommendation specifying a single textual definition of an identifier is followed the output of a program will not depend on the function chosen.

Example

In the following example the two definitions of the function `f` are different. The developer has used the fact that the call in `g` occurs in a context where the test performed in the external definition is known to be true. A simplified version of the definition of `f`, applicable to the call made in `g`, has been created and it is hoped that this will result in inline substitution.

file_1.c

```
1  void f(void)
2  {
3  if (complicated_test) /* A time consuming test. */
4     do_something;
5  else
6     do_something_else;
7  }
```

file_2.c

```
1  inline void f(void)
2  {
3  do_something;
4  }
5
6  void g(void)
7  {
8  if (part_of_complicated_test)
9     {
10    some_code;
11    if (rest_of_complicated_test)
12       f();
13    }
14 }
```

EXAMPLE inline

EXAMPLE The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit. 1544

```
inline double fahr(double t)
{
        return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
        return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double);        // creates an external definition

double convert(int is_fahr, double temp)
{
        /* A translator may perform inline substitutions */
        return is_fahr ? cels(temp) : fahr(temp);
}
```

Note that the definition of **fahr** is an external definition because **fahr** is also declared with **extern**, but the definition of **cels** is an inline definition. Because **cels** has external linkage and is referenced, an external

definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either may be used for the call.

Commentary

Although `fahr` is an external definition, an implementation may still choose to inline calls to it, from within the definition of `convert`.

C++

The declaration:

```
extern double fahr(double);     // creates an external definition
```

does not create a reference to an external definition in C++.

1545 **Forward references:** function definitions (6.9.1).

1546 120) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

footnote 120

Commentary

The only objects that can have static storage duration are those that have no linkage and are not modifiable lvalues.

1524 inline static storage duration

C++

A ***static*** *local variable in an extern inline function always refers to the same object. A string literal in an* ***extern inline*** *function is the same object in different translation units.* 7.1.2p4

The C++ Standard is silent about the case where the **extern** keyword does not appear in the declaration.

```
inline const char *saddr(void)
{
static const char name[] = "saddr";
return name;
}

int compare_name(void)
{
return saddr() == saddr(); /* may use extern definition in one case and inline in the other */
                           // They are either the same or the program is
                           // in violation of 7.1.2p2 (no diagnostic required)
}
```

6.7.5 Declarators

declarator syntax

1547

declarator:
pointer$_{opt}$ *direct-declarator*
direct-declarator:
identifier
(*declarator* **)**
direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[** **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**

```
                direct-declarator [ type-qualifier-list static assignment-expression ]
                direct-declarator [ type-qualifier-list_opt * ]
                direct-declarator ( parameter-type-list )
                direct-declarator ( identifier-list_opt )

pointer:
                * type-qualifier-list_opt
                * type-qualifier-list_opt pointer
type-qualifier-list:
                type-qualifier
                type-qualifier-list type-qualifier
parameter-type-list:
                parameter-list
                parameter-list , ...
parameter-list:
                parameter-declaration
                parameter-list , parameter-declaration
parameter-declaration:
                declaration-specifiers declarator
                declaration-specifiers abstract-declarator_opt
identifier-list:
                identifier
                identifier-list , identifier
```

Commentary

Parentheses can be used to change the way tokens in declarators are grouped in a similar way to the grouping of operands in an expression.

parameter 1603
declaration
typedef name
in parentheses

The declaration of function parameters does not have the flexibility available to declarations that are not parameters. For instance, it is not possible to write `void f(double a, b, c)` instead of `void f(double a, int b, int c)`.

C90

Support for the syntax:

```
direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
direct-declarator [ static type-qualifier-list_opt assignment-expression ]
direct-declarator [ type-qualifier-list static assignment-expression ]
direct-declarator [ type-qualifier-list_opt * ]
```

is new in C99. Also the C90 Standard only supported the form:

```
direct-declarator [ constant-expression_opt ]
```

C++

The syntax:

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[** **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{opt}$ * **]**
direct-declarator **(** *identifier-list*$_{opt}$ **)**

is not supported in C++ (although the form *direct-declarator* **[** *constant-expression*$_{opt}$ **]** is supported).

The C++ Standard also supports (among other constructions) the form:

8p4

direct-declarator **(** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

The C++ Standard also permits the comma before an ellipsis to be omitted, e.g., `int f(int a ...);`.

Other Languages

The extent to which the identifier being declared is visually separate from its associated type information varies between languages. At one extreme languages in the Pascal family completely separate the two (both Pascal and Ada require that a colon, **:**, appear between an identifier and its type information). C (and C++) are at the other extreme, integrating the identifier being declared into the syntax of the type declaration. Fortran might be considered intermediate, with the identifier being syntactically integrated with the type in some cases (e.g., array declarations `REAL A(10)`).

Common Implementations

The type qualifiers **near**, **far**, **pascal**, **tiny**, and **huge** (sometimes prefixed with one or more underscores) are ubiquitous extensions for implementations targeting the Intel x86 processor family (these qualifiers are also found in other implementations where the target processor supports a variety of pointer sizes). `gcc` supports a variety of kinds of declaration containing the keyword **__attribute__** that specify a variety of different kinds of semantic information about the identifier being declared. The HP C/iX translator[1039] supports a short pointer (32-bit) and long pointer (64-bit). The declaration of a type denoting a long pointer uses the punctuator `^`. For instance, `int ^long_ptr`.

Coding Guidelines

A very common mistake made by beginners is to treat the following two declarations of `arr` as being compatible:

1588 EXAMPLE array not pointer

```
int arr[];
```

and

```
int *arr;
```

being unaware that the duality between arrays and pointers only holds in expressions. However, this is a developer education rather than a coding guideline issue.

A more subtle mistake that even experienced developers make is to treat the star, `*`, token as belonging to the type information. For instance, in:

```
char * pc_1,
       pc_2;  /* Has type char, not char * */
char (* pc_3),
        pc_4; /* Same type as pc_2.        */
```

it is only p_1 and p_3 that have pointer types. One possible reason for the categorizations mistake is the proximity of the type specifier and the * token. Developers do not seem to make the same mistake with array bounds declarators, where an identifier appears between the type and the **[ ]** tokens. The following guideline recommendation is a simple method of avoiding this kind of mistake.

Cg 1547.1 The number of star, *, tokens appearing on each declarator of the same declarator list of a declaration shall be the same.

Redundant parentheses do not commonly appear in declarators and this issue is not discussed further.

Example

```
1  int (x); /* Redundant () */
2
3  int (((aa[1])[1])[1]); /* Harder to follow than without ()? */
4  int ab[1][1][1];
5
6  int (*ap1)[10]; /* Pointer to array of 10 ints. */
7  int *(ap2[10]); /* Array of pointers to int. */
8  int *ap3[10];   /* Which is this? */
9
10 typedef int * p_i;
11
12 p_i * p_1,
13       p_2;
```

Semantics

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers. 1548

Commentary

The form of an identifier in an expression is likely to be the same as that in the declarator. For instance, the declarator * x will have this form in an expression when the value pointed to by x is required and the declarator y[2] will have this form in an expression when an element of the array y is referred to. It is the declarator portion of a declaration that declares the identifier. There is a special kind of declarator, an *abstract-declarator*, which does not declare an identifier.

abstract declarator syntax 1624

full declarator A *full declarator* is a declarator that is not part of another declarator. 1549

Commentary

full expression 1712

This defines the term *full declarator* (which mimics that for full expression). A declarator appearing as an operand of a cast operator, that is not part of an initializer, is a full declarator. The declarators for the parameters in a function declaration are part of another declarator.

C90

Although this term was used in the C90 Standard, in translation limits, it was not explicitly defined.

C++

This term, or an equivalent one, is not defined by the C++ Standard.

full declarator sequence point The end of a full declarator is a sequence point. 1550

Commentary

It is possible for a nonconstant expression to occur within a declarator (for instance, the expression denoting the number of array elements) and its may cause side effects. In: 185 side effect

```
extern int glob;

void f(char a_1[glob++], char a_2[glob++]])
{ /* ... */ }
```

941 object modified once between sequence points

`glob` is modified twice between two adjacent sequence points and the behavior is undefined. While in:

```
int g(void)
{
static int val;
return ++val;
}

void f(char a_1[g()], char a_2[g()])
{ /* ... */ }
```

the behavior is unspecified.

C90

The ability to use an expression causing side effects in an array declarator is new in C99. Without this construct there is no need to specify a sequence point at the end of a full declarator.

C++

The C++ Standard does not specify that the end of ant declarator is a sequence point. This does not appear to result in any difference of behavior.

Other Languages

Those languages supporting some form of execution time array bounds selection invariably have the same behavior (i.e., an evaluation order is not defined).

Coding Guidelines

This ability for declarators to cause side effects is new in C99, although declarations could cause side effects in C90 through the use of an initializer. This issue of side effects is discussed elsewhere 941 object modified once between sequence points

Example

```
extern int glob;

void f(void)
{
int arr_1[glob++],
    arr_2[glob++];
}
```

1551 If in the nested sequence of declarators in a full declarator ~~contains~~ there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. variably modified

Commentary

This defines the term *variably modified*. The parameter types or return type of a function type are not usually considered to be *nested* within its full declarator.

The wording was changed by the response to DR #311.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified. 1552

Commentary

This wording ensures that in code such as the following example, the declaration of y is also variably modified.

```
1  int x;
2
3  typedef int vla[x];
4  vla y[3];
```

This sentence was added by the response to DR #311.

In the following subclauses, consider a declaration 1553

T D1

where **T** contains the declaration specifiers that specify a type *T* (such as **int**) and **D1** is a declarator that contains an identifier *ident*.

Commentary

This is the simplest, most basic, form of declaration (the various possible declarators, D1, supported in C, are described in the following C sentences).

The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation. 1554

Commentary

abstract declarator syntax 1624

In the case of abstract declarators there is no identifier (although discussions involving these constructs sometimes refer to an *omitted* identifier).

If, in the declaration "**T D1**", **D1** has the form 1555

identifier

then the type specified for *ident* is *T*.

Commentary

This is the most common form of declaration, declaring an identifier to have type T.

If, in the declaration "**T D1**", **D1** has the form 1556

(D)

then *ident* has the type specified by the declaration "**T D**".

Commentary

That is, the use of parentheses is purely a syntactic device that may affect the parsing of a sequence of tokens.

1557 Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Commentary

This specifies that this parenthesis around declarators have no semantics associated with them (they have associated semantics when they appear to the right of an identifier, they cause it to denote a function type). It is also possible to use typedefs to achieve the same effect. For instance:

```
int (*V_1)[10]; /* Pointer to array of int. */

typedef int A10_INT[10];
A10_INT *V_2;   /* Pointer to array of int. */
```

Other Languages

Many languages do not support the bracketing of identifiers in declarations. It is only needed in C because the syntax of declarators uses tokens that can appear to the left or the right of the identifier whose type they specify. To specify some types parentheses are needed to alter the binding. For instance,

```
int *V[10];   /* Array of pointer to int. */
int (*V)[10]; /* Pointer to array of int. */

/*
 * Type information is Pascal always reads left-to-right.
 */
V : array[0..9] of ^ integer; /* Array of pointer to int. */
V : ^array[0..9] of integer;  /* Pointer to array of int. */
```

Common Implementations

Parentheses are processed as part of the syntax. Their presence can alter the parsing (binding) of declarators. There are no implementation issues associated by reordering declarators through parenthesis, like there are for expressions.

981 parenthesized expression

Coding Guidelines

Experience shows that declarators of the form `*x[3]` are a source of developer miscomprehension. Some developers reading it left-to-right "a pointer to an array of . . . ", rather than the correct right-to-left order "array of pointer to . . . ". The interpretation of an objects type affects how it is used in expressions. Depending on how x is accessed the final operand type may become "a pointer to a pointer to . . . " under both interpretations of the type of x (indexing an object having an array type causes it to be converted to a pointer type in many contexts). A consequence of this conversion is that it is possible for x to be accessed, under both interpretations of its type, without a translator diagnostic (pointing out a type mismatch) being issued.

729 array converted to pointer

If the guideline recommendation specifying the use of parenthesis in expressions is applied to declarations, a reader of the source may notice a discrepancy between their incorrect interpretation of the type specified in an objects declaration and the type implied by how that object is used in an expression. However, using parenthesis to unambiguously define the intended syntactic binding of the components of a declarator is a more reliable method of ensuring that readers comprehend the intended type.

943.1 expression shall be parenthesized

Cg 1557.1 The declarator for an object declared to have type "array of pointer to . . . " shall parenthesize the array portion of the declarator.

Dev 1557.1 The array portion of a declarator need not be parenthesized if its element type is specified using a typedef name.

Example

```
#define INT_PTR int *
typedef int * int_ptr;
int * arr_1  [3];
int (*arr_2) [3]; /* Parenthesis must be used to declare this type. */
int *(arr_3  [3]);
int_ptr arr_4[3]; /* Unlikely to be interpreted as a pointer to an array. */
INT_PTR arr_5[3]; /* Not covered by previous deviation because the usage is recommended against elsewhere. *
```

Implementation limits

declarator complexity limits

As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or incomplete type, either directly or via one or more **typedef**s. 1558

Commentary

limit 279 type complexity

This wording differs from that in 5.2.4.1 in that it includes types defined via typedefs in the count.

C90

The implementation shall allow the specification of types that have at least 12 pointer, array, and function declarators (in any valid combinations) modifying an arithmetic, a structure, a union, or an incomplete type, either directly or via one or more ***typedef****s.*

Forward references: array declarators (6.7.5.2), type definitions (6.7.7). 1559

6.7.5.1 Pointer declarators

Semantics

derived-declarator-type-list

If, in the declaration "**T D1**", **D1** has the form 1560

```
* type-qualifier-list_opt D
```

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*".

Commentary

The term *derived-declarator-type-list* is very rarely used outside of the C Standard, even during Committee discussions.

C++

The C++ Standard uses the term *cv-qualifier-seq* instead of *type-qualifier-list*.

For each type qualifier in the list, *ident* is a so-qualified pointer. 1561

Commentary

The C++ Standard says it better:

8.3.1p1

The `cv-qualifiers` *apply to the pointer and not to the object pointed to.*

So qualifiers to the right of the * token qualify the pointer type and qualifiers to the left of the * token qualify the pointed-to type.

Example

The following declares three objects to have the same type.

```
typedef int * p_i;

p_i   const g_1;
const p_i   g_2;
const int  *g_3;
```

1562 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

pointer types to be compatible

Commentary

There is no requirement that the declarators of the two pointer types, being checked for compatibility, be built from the same sequence of tokens. For instance:

631 compatible type if

```
typedef int * int_ptr;

int *   p1;
int_ptr p2; /* The type of p1 is compatible with that of p2. */
```

C++

The C++ Standard does not define the term *compatible type*, although in the case of qualified pointer types the term *similar* is defined (4.4p4). When two pointer types need to interact the C++ Standard usually specifies that the qualification conversions (clause 4) are applied and then requires that the types be the same. These C++ issues are discussed in the sentences in which they occur.

1563 EXAMPLE The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value".

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by **`ptr_to_constant`** shall not be modified through that pointer, but **`ptr_to_constant`** itself may be changed to point to another object. Similarly, the contents of the **`int`** pointed to by **`constant_ptr`** may be modified, but **`constant_ptr`** itself shall always point to the same location.
The declaration of the constant pointer **`constant_ptr`** may be clarified by including a definition for the type "pointer to **`int`**".

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares **`constant_ptr`** as an object that has type "const-qualified pointer to **`int`**".

Commentary

A typename can be qualified with a type qualifier (e.g., `const int_ptr`), but the underlying type cannot be changed by adding a type specifier (e.g., `unsigned int_ptr`)

1378 type specifier syntax

6.7.5.2 Array declarators

Constraints

In addition to optional type qualifiers and the keyword **static**, the **[** and **]** may delimit an expression or *. 1564

Commentary

Saying in words what is specified in the syntax (and in a constraint!)

C90

*The expression delimited by **[** and **]** (which specifies the size of an array) shall be an integral constant expression that has a value greater than zero.*

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and support for * between **[** and **]** in a declarator is new in C99.

C++

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and * between **[** and **]** in a declarator is new in C99 and is not specified in the C++ Standard.

If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. 1565

Commentary

The expression is usually thought of, by developers, in terms of specifying the number of elements, not the size.

Other Languages

Many languages require the value of the expression to be known at translation time, and some (e.g., Ada) allow execution time evaluation of the array bounds, while a few (e.g., APL, Perl, and Common Lisp) support dynamically resizeable arrays. In some languages (e.g., Fortran) there is an implied lower bound of one. The value given in the array declaration is the upper bound and equals the number of elements. Languages in the Pascal family require that a type be given. The minimum and maximum values of this type specifying the lower and upper bounds of the array. This type also denotes the type of the expression that must be used to index that array. For instance, in:

```
C_A :Array[Char] of Integer;
```

the array C_A can only be indexed with an expression having type **char**.

Coding Guidelines

At the time of this writing there is insufficient experience with use of this construct to known whether any guideline recommendation (e.g., on the appearance of side effects) might be worthwhile.

array declaration size greater than zero

If the expression is a constant expression, it shall have a value greater than zero. 1566

Commentary

C does not support the creation of named objects occupying zero bytes of storage.

Note that this constraint is applied before any implicit conversions of array types to pointer types (e.g., even although the eventual type of a in `extern int f(int a[-1]);` is pointer to **int**, a constraint violation still occurs).

Other Languages

Very few languages support the declaration of zero sized objects (Algol 68 allows the declaration of a flexible array variable to have zero size, assigning an array to such a variable also resulting in the new bounds being assigned). Languages in the Pascal family support array bounds less than one. For instance, the declaration:

```
C_A :Array[-7..0] of Integer;
```

creates an array object, C_A, that must be indexed with values between -7 and 0 (inclusive).

1567 The element type shall not be an incomplete or function type.

array element not incomplete type
array element not function type

Commentary

The benefits of allowing developers to declare arrays having element types that are incomplete types was not considered, by the C committee, to be worth the complications (i.e., cost) created for translator writers (the C language definition is intended to translatable in a single pass). C does not support treating functions as data, so while pointers to functions are permitted arrays of function types are not.

10 implementation single pass

C90

In C90 this wording did not appear within a Constraints clause. The requirement for the element type to be an object type appeared in the description of array types, which made violation of this requirement to be undefined behavior. The undefined behavior of all known implementations was to issue a diagnostic, so no actual difference in behavior between C90 and C99 is likely to occur.

526 array contiguously allocated set of objects

C++

8.3.4p1

T is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type **`void`***, a function type or an abstract class type.*

The C++ Standard does not disallow incomplete element types (apart from the type **`void`**). This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

Other Languages

Few languages support arrays of functions (or pointers to functions).

1568 The optional type qualifiers and the keyword **`static`** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

array parameter qualifier only in outermost

Commentary

The functionality provided by the appearance of type qualifiers and the keyword **`static`** in this context is not needed for declarations involving array types in other contexts (because function parameters are the only context where the declaration of an object having an array type is implicitly converted to a pointer type).

729 array converted to pointer

C90

Support for use of type qualifiers and the keyword **`static`** in this context is new in C99.

C++

Support for use of type qualifiers and the keyword **`static`** in this context is new in C99 is not supported in C++.

Coding Guidelines

Support for these constructs is new in C99 and insufficient experience has been gained with their usage to know if any guideline recommendations are worthwhile.

1569 ~~Only an~~ An ordinary identifier (as defined in 6.2.3) ~~with both block scope or function prototype scope and no linkage shall have a variably modified type.~~ that has a variably modified type shall have either block scope and no linkage or function prototype scope.

variable modified only scope

Commentary

Ordinary identifiers do not include members of structure or union types. These members are prevented from having a variably modified type for practical implementation reasons. For instance, requirement on the relative ordering of storage allocated to members of a structure cannot be met without reserving impractically

444 ordinary identifiers

1206 structure members later compare later

large amounts of storage for any member having a variably modified type (variably modified types are usually implemented via a pointer to the actual, variable sized, storage allocated).

VLA 464
lifetime starts/ends
static storage 456
duration
when initialized

Storage for objects having external or internal linkage is only allocated once and it makes no sense for such objects to have a variably modified type.

This wording was changed by the response to DR #320.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

If an identifier is declared to be an object with static storage duration, it shall not have a variable length array type. 1570

Commentary

static storage 456
duration
when initialized

The standard requires that storage for all objects having static storage duration be allocated during program startup. Thus the amount of storage to allocate must be known prior to the start of program execution, ruling out the possibility of execution time specification of the length of an array.

Semantics

---

qualified array of

If, in the declaration "**T D1**", **D1** has one of the forms: 1571

D[*type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
D[static *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
D[*type-qualifier-list* **static** *assignment-expression* **]**
D[*type-qualifier-list*$_{opt}$ ***** **]**

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[121)]

Commentary

derived-declarator-type-list 1560

Phrases such as *derived-declarator-type-list* are rarely used outside of the C standard, even amongst the Committee.

C90

Support for forms other than:

D[*constant-expression*$_{opt}$ **]**

is new in C99.

C++

The C++ Standard only supports the form:

D[*constant-expression*$_{opt}$ **]**

A C++ translator will issue a diagnostic if it encounters anything other than a constant expression between the **[** and **]** tokens.

The type of the array is also slightly different in C++, which include the number of elements in the type:

8.3.4p1 *. . . the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T."*

Other Languages

Some languages (e.g., the Pascal family of languages) require both a lower and upper bound to be specified. Although BCPL uses the tokens **[ ]** to denote an array access (and also a function call), these tokens are not used in an array declaration. For instance, the following creates an array of 10 elements (lower bound of zero, upper bound of nine) and assigns a pointer to the first element to `N`:

```
1  let N = vec 9
```

1572 (See 6.7.5.3 for the meaning of the optional type qualifiers and the keyword **static**.)

Commentary

See elsewhere for commentary on optional type specifiers and the keyword **static**.

1598 array type
adjust to pointer to
1599 function declarator
static
array
incomplete type

1573 If the size is not present, the array type is an incomplete type.

Commentary

This is stated is a different way elsewhere. As well as appearing between **[]** tokens, the size may also be specified, in the definition of an object, via the contents of an initializer.

546 array
unknown size
1683 array of
unknown size
initialized

C++

The C++ Standard classifies all compound types as object types. It uses the term *incomplete object type* to refer to this kind of type.

475 object types

> *If the constant expression is omitted, the type of the identifier of* `D` *is "*`derived-declarator-type-list` *array of unknown bound of* `T`*," an incomplete object type.* 8.3.4p1

Coding Guidelines

The following are some of the contexts in which the size might not be present:

1683 array of
unknown size
initialized

- An automatically generated initializer, used in an object definition, may be written to a file that is **#include**d at the point of use (while the generation of the members of the initializer may be straightforward, the cost of automatically modifying the C source code, so that it contains an explicit value for the number of elements may be not be considered worth the benefit).
- A typedef name denoting an array type of unknown size is a useful way of giving a name to an array of some element type (the size being specified later when an object, of that named type, is defined).
- The type of a parameter in a function declaration. While this usage may seem natural to developers who have recently moved to C from some other languages, it looks unusual to experienced C developers (the parameter is not treated as an array at all, its type is converted to a pointer to the element type).

729 array
converted to
pointer

- The type of an object declared in a header, where there is a desire to minimize the amount of visible information. In this case a pointer could serve the same purpose. The only differences are in how the storage for the object is allocated (and initialized) and the machine code generated for accesses (a pointer requires an indirect load, a minor efficiency penalty; the use of an array can simplify an optimizers attempt to deduce which objects are not aliased, potentially leading to higher-quality machine code. The introduction of the keyword **restrict** in C99 has potentially removed this advantage).

While not having the number of elements explicitly specified in the declaration of an array type may be surprising to users of other languages (experienced users of other languages, sometimes promoted to a level whether they no longer write code and are not experienced with C, are surprisingly often involved in the creation of a company's coding guidelines) the usage is either cost effective or the alternatives offer no additional benefits. For these reasons no guideline recommendation is given here.

Example

In the following:

```
extern void f(int *p1, int *p2);
extern void f(int p1[], int p2[3]);
```

the first parameter is not an array of known size in either declaration, but the types are compatible. Possible composite types are:

```
extern void f(int *p1, int p2[3]);
```

and

```
extern void f(int p1[], int p2[3]);
```

In the following:

```
extern int g(int p3[3]);
extern int g(int p3[5]);
```

the parameter types are compatible (because both are first converted to pointers before checking for compatibility). Possible composite types are:

```
extern int g(int p3[3]);
```

and

```
extern int g(int p3[5]);
```

In the following:

```
extern int h(int n, int p4[2]);
extern int h(int n, int p4[n]);
```

The composite type is:

```
extern int h(int p4[2]);
```

---

variable length array specified by *

If the size is * instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope;[122)] 1574

Commentary

It is possible that implementations will want to use different argument passing conventions for variable length array types than for other array types. The * notation allows a variable length array type to be specified without having to specify the expression denoting its size, removing the need for the identifiers used in the size expression to be visible at the point in the source the function prototype is declared. If an expression is given in function prototype scope, it is treated as if it were replaced by *.

VLA 1581 size treated as *

Syntactically the size can be specified using * in any context that an array declarator can appear. If this context is not in function prototype scope the behavior is undefined.

C90

Support for a size specified using the * token is new in C99.

C++

Specifying a size using the * token is new in C99 and is not available in C++.

Other Languages

Several languages (e.g., Fortran) use * to denote an array having an unknown number of elements.

1575 such arrays are nonetheless complete types.

Commentary

The size of a parameters type in function prototype scope is not needed by a translator. Specifying that arrays declared using the * notation are complete types enables pointers to variable length arrays, and arrays having more than one variable length dimension, to be declared as parameters, e.g., `int f(int p_1[*][*], int (*p_2)[*])`.

1576 If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type;

variable length array type

Commentary

This defines a variable length array type in terms of what it is not. In:

548 known constant size

```
extern int n,
           m;
int a_1[5][n]; /* Size is not an integer constant expression. */
int a_2[m][6]; /* Element type is not a known constant size.  */
```

C90

Support for specifying a size that is not an integer constant expression is new in C99.

C++

Support for specifying a size that is not an integer constant expression is new in C99 and is not specified in the C++ Standard.

1577 121) When several "array of" specifications are adjacent, a multidimensional array is declared.

footnote 121

Commentary

Although not notated as such, this is the definition of the term *multidimensional array*.

Other Languages

The process of repeating array declarators to create multidimensional arrays is used in many languages. Some languages (e.g., those in the Algol family) support the notational short cut of allowing information on all dimensions to occur within one pair of brackets (when this short cut is used it is not usually possible to access slices of the array). For instance, a multidimensional array declaration in Pascal could be written using either of the following forms:

```
Arr_1 : Array[1..4] of Array[0..3] of Integer;
Arr_2 : Array[1..4, 0..3] of Integer;
```

while in Fortran it would be written:

```
INTEGER Arr_2(10,20)
```

Common Implementations

To improve performance, when accessing multidimensional arrays, Larsen, Witchel, and Amarasinghe[812] found that it was sometimes worthwhile to pad the lowest dimension array to ensure the next higher dimension started at a particular alignment (whole program analysis was used to verify that there was no danger of changing the behavior of the program; multidimensional arrays are required to have any no padding between dimensions). For instance, if the lowest dimension was specified to contain three elements, it might be worthwhile to increase this to four.

footnote 122

122) Thus, * can be used only in function declarations that are not definitions (see 6.7.5.3). 1578

Commentary

object 1361 type complete by end

The size of all parameters in a function definition are required to be known (i.e., they have complete types), even if the parameter is only passed as an argument in a call to another function.

array variable length

otherwise, the array type is a variable length array type. 1579

Commentary

This is true by definition.

If the size is an expression that is not an integer constant expression: 1580

Commentary

integer constant expression 1328

That is, the expression does not have a form that a translator is required to be able to evaluate to an integer constant during translation.

C90

Support for non integer constant expressions in this context is new in C99.

C++

Support for non integer constant expressions in this context is new in C99 and is not available in C++.

VLA size treated as *

if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by *; 1581

Commentary

The implication of this wording is that the expression is not evaluated. A size expression occurring in a declaration at function prototype scope serves no purpose other than to indicate that the parameter type is a variably modified array type.

variable length array specified by * 1574

Other Languages

Languages that support execution time evaluation of an arrays size face the same problems as C. Few checks can be made on the type during translation and the expression essentially has to be treated as evaluating to some unknown value.

Coding Guidelines

The expression might be given for documentation purposes. It provides another means for developers to obtain information about the expected size of the array passed as an argument. Whether such usage is more likely to be kept in sync with the actual definition than information given in a comment is not known. A guideline recommendation that the expression be identical to that in the function definition is not given for the same reason that none is given for keeping comments up-to-date with the code.

comment disadvantages

otherwise, each time it is evaluated it shall have a value greater than zero. 1582

Commentary

array declaration size greater than zero 1566

This specification is consistent with that given when the expression is a constant. The expression is evaluated each time the function is called, even if the call is a recursive one.

Example

```
1  #include <stddef.h>
2
3  size_t fib(size_t n_elems, char arr[static n_elems],
4             char p_a[static (n_elems == 1) ? 1 : (n_elems + fib(n_elems-1, arr, arr))])
```

```
5   {
6   return sizeof(p_a);
7   }
```

1583 The size of each instance of a variable length array type does not change during its lifetime.

Commentary

The implication is that subsequent changes to the values of objects appearing in the expression do not affect to size of the VLA. The size expression is evaluated only when the declarator containing it is encountered during program execution. The number of elements in the array type is then fixed throughout its lifetime. In the following the change in the value of `glob` does not array the value of `sizeof(arr)`.

1711 object initializer evaluated when
1841 function entry parameter type evaluated

```
1   extern int glob;
2
3   void f(void)
4   {
5   int arr[glob];
6
7   glob++;
8   sizeof(arr);
9   }
```

A variable length array type need not denote a named object. For instance, in:

```
1   void f(int n)
2   {
3   static char (*p)[n++];
4   }
```

the pointer `p` has a variably modified type, but is not itself a VLA. Even although storage for `p` is allocated at program startup its type is not known until the array declarator is evaluated, each time `f` is called (any assignment to `p` has to be compatible with this type).

Other Languages

A few languages (e.g., APL and Perl) support arrays that change their size on an as-needed basis.

Coding Guidelines

It is possible that the expression denoting the size of the VLA array will also appear elsewhere in the function, e.g., to perform some array bounds test. Any modification of the value of an object appearing in this expression is likely to cause the value of the two expressions to differ. Until more experience is gained with the use of VLA types it is not possible to evaluate the cost/benefit of any potential guideline recommendations (e.g., recommending against the modification of objects appearing in the expression denoting the size of a VLA).

1584 Where a size expression is part of the operand of a **`sizeof`** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated.

sizeof VLA unspecified evaluation

Commentary

The rather poor argument (in your authors opinion) put forward, by some Committee members, for this unspecified behavior, was that there were existing implementations that did not evaluate sub-operands that did not affect the value returned by the **`sizeof`** operator.

C90

The operand of **`sizeof`** was not evaluated in C90. With the introduction of variable length arrays it is possible that the operand will need to be evaluated in C99.

Common Implementations

As part of the process of simplifying and folding expressions some translators only analyze as much of their intermediate representation as is needed. Such translators are unlikely to generate machine code to evaluate any operands of the **sizeof** operator that are not needed to obtain the value of an expression.

Coding Guidelines

Evaluation of the operands of **sizeof** only becomes an issue when it may appear to generate a side effect. However, no guideline recommendation is given here for the same reason that none was given for the non-VLA case.

sizeof 1123 operand not evaluated

Example

```
1  extern int glob;
2
3  void f(int n,
4         int arr[++n]) /* Expression evaluated. */
5  {
6  int (*p)[glob++];    /* Expression evaluated. */
7  /*
8   * To obtain the size of the type ++n must be evaluated,
9   * but the value of ++glob does not affect the final size.
10  */
11 n=sizeof(int * (int * [++glob])[++n]);
12 }
```

---

array type to be compatible

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. 1585

Commentary

structural compatibility

structural 1585 compatibility
compatible 631 type if

While the compatibility rules for structure and union types are based on the names of the types, array type compatibility is based on what is sometimes called *structural compatibility*. The components of the type, rather than just its name, is compared. This requirement can be verified at translation time. If either size specifier is not present then the array types are always compatible (provide their element types are).

C++

compatible 631 type if

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

Other Languages

Languages in the Pascal family usually compare the names of array types, rather than their components. This is sometimes considered too inconvenient when passing arguments having an array type and special rules for argument/parameter array type compatibility are sometimes created (e.g., conformant arrays in Pascal). Some languages (e.g., CHILL) use *structural compatibility* type checking rules. When the lower bound of an array is specified by the developer it is necessary to check that both the lower and upper bounds, for each dimension of the two arrays, have the same value.

If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values. 1586

Commentary

array 729 converted to pointer

In most contexts expressions having an array type are converted to a pointer to their element type. Contexts where the size specifiers need to be considered include the following:

function 1599 declarator static

- an argument whose corresponding parameter has an array type that includes the type qualifier **static**, and

- assignment to an object having type pointer to array of type.

1288 assignment-expression syntax

C90

The number of contexts in which array types can be incompatible has increased in C99, but the behavior is intended to be the same.

C++

There are no cases where the C++ Standard discusses a requirement that two arrays have the same number of elements.

Common Implementations

In practice the unexpected (undefined) behavior may not occur after the point of assignment. For instance, at the point where an object having the array type is accessed. For arrays having a single dimension, provided the access is within the bounds of the object, an implementation is likely to behave the same way as when the two array types have equal size specifiers. For multidimensional arrays the number of elements in each array declarator, except the last one, is used as a multiplier for calculating the index value. This means that even if the object, having an array type, is larger than required the incorrect element will still be accessed.

```
1  static int glob = 9;
2
3  void f(char p_arr[static 4][static 6])
4  {
5  /*
6   * The following will access location p_arr+(2+3*4) had the sizes
7   * been the same location p_arr+(2+3*9) would have been accessed.
8   */
9  p_arr[2][3] = 9;
10 }
11
12 void g(void)
13 {
14 char arr[glob][4];
15
16 f(arr);
17 }
```

Coding Guidelines

Developers may intentionally write code where the two array specifiers have different values, relying on the behavior being defined when only allocated storage is accessed. Support for VLA types are new in C99 and there is not yet sufficient experience to be able to judge the cost/benefit of recommending against this usage. Developers may also accidentally write code where two array specifiers have different values. However, these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

0 guidelines not faults

1587 EXAMPLE 1

EXAMPLE array of pointers

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

Commentary

The algorithm for reading declarations involving both array and pointer types is to:

1. start at the identifier and work right through each array size, until a semicolon or closing parenthesis is reached,
2. then restart at the identifier and work left through the type qualifiers and *'s,

3. if a closing parenthesis is reached on step 1, it the opening parenthesis will eventually be reached on step 2. This parentheses bracketed sequence is then treated as an identifier and the process repeated from step 1.

EXAMPLE function returning pointer to 1617 A similar rule can be applied to reading function types.

Coding Guidelines

symbolic name 822 The discussion on using symbolic names rather than integer constants is applicable here.

Example

```
int *   (*   (*glob[1]      [2])     [3])     [4];
/*
                    array of
                             array of
               pointer to
                                      array of
          pointer to
                                               array of
     pointer to

or in linear form:

array of array of pointer to array of pointer to array of pointer to int
*/
```

EXAMPLE array not pointer EXAMPLE 2 Note the distinction between the declarations 1588

```
extern int *x;
extern int y[];
```

The first declares **x** to be a pointer to **int**; the second declares **y** to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

Commentary

declarator syntax 1547 The issue of the same object being declared to have both pointer and array types, one in each of two translation units, is discussed elsewhere.

EXAMPLE 3 The following declarations demonstrate the compatibility rules for variably modified types. 1589

```
extern int n;
extern int m;
void fcompat(void)
{
        int a[n][6][m];
        int (*p)[4][n+1];
        int c[n][n][6][m];
        int (*r)[n][n][n+1];
        p = a;        // invalid: not compatible because 4 != 6
        r = c;        // compatible, but defined behavior only if
                      // n == 6 and m == n+1
}
```

Commentary

Array types having more than two dimensions are rarely seen in practice (see Table 991.1).

Other Languages

In some languages (e.g., APL, Perl) assigning a value may change the type of the object assigned to (to be that of the value assigned).

1590 EXAMPLE 4 All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **static** or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                           // invalid: file scope VLA
extern int (*p2)[n];                // invalid: file scope VM
int B[100];                         // valid: file scope but not VM

void fvla(int m, int C[m][m]);      // valid: VLA with prototype scope

void fvla(int m, int C[m][m])       // valid: adjusted to auto pointer to VLA
{
        typedef int VLA[m][m];      // valid: block scope typedef VLA

        struct tag {
                int (*y)[n];        // invalid: y not ordinary identifier
                int z[n];           // invalid: z not ordinary identifier
        };
        int D[m];                   // valid: auto VLA
        static int E[m];            // invalid: static block scope VLA
        extern int F[m];            // invalid: F has linkage and is VLA
        int (*s)[m];                // valid: auto pointer to VLA
        extern int (*r)[m];         // invalid: r has linkage and points to VLA
        static int (*q)[m] = &B;    // valid: q is a static block pointer to VLA
}
```

Commentary

In this example `m` is being used in the same way as a symbolic name might be used in the declaration of non-VM array types.

1591 **Forward references:** function declarators (6.7.5.3), function definitions (6.9.1), initialization (6.7.8).

6.7.5.3 Function declarators (including prototypes)

Constraints

1592 A function declarator shall not specify a return type that is a function type or an array type.

function declarator return type

Commentary

The wording given for function definitions is slightly different and the discussion given for that sentence is applicable here.

1823 function definition return type

Other Languages

It is quite common for languages to support functions returning array types. Support for functions returning function types is much less common (e.g., Lisp, Algol 68).

1593 The only storage-class specifier that shall occur in a parameter declaration is **register**.

parameter storage-class

Commentary

Parameters have block scope and automatic storage duration. Like any other object having block scope the developer might want to suggest that accesses to it be optimized. Use of any other storage-class specifier in a parameter declaration has no obvious semantics.

408 block scope terminates
457 automatic storage duration
1369 register storage-class

C++

7.1.1p2 *The* **auto** *or* **register** *specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).*

C source code developed using a C++ translator may contain the storage-class specifier **auto** applied to a parameter. However, usage of this keyword is rare (see Table 788.1) and in practice it is very unlikely to occur in this context.

The C++ Standard covers the other two cases with rather sloppy wording.

7.1.1p4 *There can be no* **static** *function declarations within a block, nor any* **static** *function parameters.*

7.1.1p5 *The* **extern** *specifier cannot be used in the declaration of class members or function parameters.*

An identifier list in a function declarator that is not part of a definition of that function shall be empty. 1594

Commentary

An identifier list provides little useful information in a function declaration that is not also a definition. The issue of an empty identifier list is discussed elsewhere.

function declarator empty list 1608

C++

The C++ Standard does not support the old style of C function declarations.

Example

```
1  extern int f_1(x, y); /* Constraint violation. */
2
3  int (*f_2(a, b))(x, y) /* The subject of an outstanding DR. */
4  { /* ... */ }
```

---

parameter adjustment in definition

After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type. 1595

Commentary

Adjustment refers to array and function types, which are implicitly converted to a pointer to their first element (there is no such conversion for incomplete structure or union types). After adjustment, the types of parameters have the same requirements as the types of other block scope objects.

array type adjust to pointer to 1598

C translators need to know the number of bytes of storage needed by each parameter. Incomplete types do not have a known, at the point of declaration, size. Arguments having an array type are passed as the address of their first element, so their size is not needed.

size needed 1465

C90

The C90 Standard did not explicitly specify that the check on the parameter type being incomplete occurred "after adjustment".

C++

The C++ Standard allows a few exceptions to the general C requirement:

8.3.5p6

If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.[87)]

This excludes parameters of type "`ptr-arr-seq` T2" where T2 is "pointer to array of unknown bound of T" and where `ptr-arr-seq` means any sequence of "pointer to" and "array of" derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc. Footnote 87

*The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case, **`void`** shall not be a parameter type (though types derived from **`void`**, such as `void*`, can).* 8.3.5p2

The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type. 8.3.5p6

```
1  void f(struct s1_tag ** p1) /* incomplete type, constraint violation */
2                              // defined behavior
3  {
4  struct s2_tag **loc; /* Size not needed, so permitted. */
5  }
```

Semantics

1596 If, in the declaration "**`T D1`**", **`D1`** has the form

`D(` *parameter-type-list* **`)`**

or

`D(` *identifier-list*$_{opt}$ **`)`**

and the type specified for *ident* in the declaration "**`T D`**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning *T*".

Commentary

This defines the terms *derived-declarator-type-list* and *derived-declarator-type-list*. They are very rarely used outside of the standard, even by the Committee.

C++

The form supported by the C++ Standard is:

D1 **`(`** *parameter-declaration-clause* **`)`** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$ 8.3.5p1

The term used for the identifier in the C++ Standard is:

8.3.5p1

"derived-declarator-type-list function of (parameter-declaration-clause) cv-qualifier-seq^{opt} returning T";

The old-style function definition *D(identifier-list$_{opt}$)* is not supported in C++.

8.3.5p2 *If the parameter-declaration-list is empty, the function takes no arguments. The parameter list (void) is equivalent to the empty parameter list.*

The C syntax treats D() as an instance of an empty identifier-list, while the C++ syntax treats it as an empty *parameter-type-list*. Using a C++ translator to translate source containing this form of function declaration may result a diagnostic being generated when the declared function is called (if it specifies any arguments).

Coding Guidelines

function declaration use prototype 1810.1 prototypes 1810 cost/benefit

If the guideline recommendation specifying the use of prototypes is followed the identifier list form will not occur in new source code. However, it may occur in existing source code and the cost/benefit issues associated with changing this existing usage are discussed elsewhere.

A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. 1597

Commentary

Rationale There was considerable debate about whether to maintain the current lexical ordering rules for variable length array parameters in function definitions. For example, the following old-style declaration

```
void f(double a[*][*], int n);

void f(a, n)
    int n;
    double a[n][n];
{
    // ...
}
```

cannot be expressed with a definition that has a parameter type list as in

```
void f(double a[n][n], int n)  // error
{
    /* ... */
}
```

Previously, programmers did not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. With Standard C's lexical ordering rules, the declaration of a would force n to be undefined or captured by an outside declaration. The possibility of allowing the scope of parameter n to extend to the beginning of the parameter-type-list was explored (relaxed lexical ordering), which would allow the size of parameter a to be defined in terms of parameter n, and could help convert a Fortran library routine into a C function. Such a change to the lexical ordering rules is not considered to be in the "Spirit of C", however. This is an unforeseen side effect of Standard C prototype syntax.

C++

The `parameter-declaration-clause` *determines the arguments that can be specified, and their processing, when the function is called.*

Other Languages

Many languages allow parameters to be defined to have types in the same way that they support the definition of objects. Some languages provide no explicit mechanism for defining the types of the parameters (e.g., Perl).

Coding Guidelines

Some coding guideline documents recommend that no identifiers be given for the parameters in a function prototype declaration. The rationale is that such identifiers could also match some earlier defined macro name. In practice, the effect of such a match is likely to be a syntax violation. Even if this does not occur the spelling of the identifier appearing in the prototype declaration is not significant. Whatever happens there will not be a quiet change in program behavior. The cost of a syntax violation (the identifier spelling will need to be changed) has to be balanced against the benefit of including an identifier in the parameter type list.

Some coding guideline documents recommend that any identifiers given for the parameters in a function prototype declaration have the same spelling as those given in the function definition. Such usage may provide readers with a reminder of information about what the parameter denotes. A comment could provide more specific information. Using identifiers in this way also provides visible information that might help detect changes to a functions interface (e.g., a change in the order of the parameters).

There does not appear to be compelling evidence for any of these options providing sufficient cost/benefit for a guideline recommendation to be worthwhile.

Example

```
#define abc xyz

void f_1(int abc);

int cost_weight_ratio(int cost, int weight);

int cost_weight_ratio(int weight, int cost)
{
return cost / weight;
}
```

1598 A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation.

array type adjust to pointer to

Commentary

This is a requirement on the implementation.

```
void f(      int a1[10],       /* equivalent to       int *       a1 */
       const int a2[10],       /* equivalent to const int *       a2 */
             int a3[const 10], /* equivalent to       int * const a3 */
       const int a4[const 10]) /* equivalent to const int * const a4 */
{ /* ... */ }
```

Occurrences of an object, not declared as a parameter, having an array type are implicitly converted to a pointer type in most contexts. The parameter declaration in:

729 array converted to pointer

```
void f(int a[const 10][const 20])
{ /* ... */ }
```

is permitted by the syntax, but violates a constraint.

1568 array parameter qualifier only in outermost

C90

Support for type qualifiers between **[** and **]**, and the consequences of their use, is new in C99.

C++

This adjustment is performed in C++ (8.3.5p3) but the standard does not support the appearance of type qualifiers between **[** and **]**.

Source containing type qualifiers between **[** and **]** will cause a C++ translator to generate a diagnostic.

Other Languages

Using qualifiers within the **[** and **]** of an array declaration may be unique to C.

Coding Guidelines

A qualifier appearing outside of **[** and **]** qualifies the element type, not the array type. For non-parameter declarations this distinction is not significant, the possible consequences are the same. However, the implicit conversion to pointer type, that occurs for parameters having an array type, means that the distinction is significant in this case. Experience shows that developers are not always aware of the consequences of this adjustment to parameters having an array type. The following are two of the consequences of using a qualifier in the incorrect belief that the array type, rather than the element type, will be qualified:

- The **volatile** qualifier— the final effect is very likely to be the intended effect (wanting to **volatile** qualify an object having a pointer type is much rarer than applying such a qualifier to the object it points at).
- The **const** qualifier— attempts to modify the pointed-to objects will cause a translator diagnostic to be issued and attempts to modify the parameter itself does not require a translator to issue a diagnostic.

Support for qualifiers appearing between **[** and **]** is new in C99 and there is insufficient experience in their use to know whether any guideline recommendation is cost effective.

---

function declarator static

If the keyword **static** also appears within the **[** and **]** of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression. 1599

Commentary

Rationale

It would be a significant advantage on some systems for the translator to initiate, at the beginning of the function, prefetches or loads of the arrays that will be referenced through the parameters.

The use of the keyword **static** within the **[** and **]** of an array type is a guarantee from the developer to the translator. The translator can assume that at least the value of the parameter will not be NULL and that storage for at least the specified number of elements will be available.

Rules for forming the composite type of function types, one or more of which included the keyword **static**, were given by the response to DR #237.

DR #237

The effect is as if all of the declarations had used static and the largest size value used by any of them. Each declaration imposes requirements on all calls to the function in the program; the only way to meet all of these requirements is to always provide pointers to as many objects as the largest such value requires.

```
1  void WG14_DR_237(int x[static 10]);
2  void WG14_DR_237(int x[static 5]);
3
4  void WG14_DR_237(int x[1])
5  /*
6   * Composite type is void WG14_DR_237(int x[static 10])
7   */
8  { /* ... */ }
```

C90

Support for the keyword **static** in this context is new in C99.

C++

Support for the keyword **static** in this context is new in C99 and is not available in C++.

Other Languages

Most strongly typed languages require an exact correspondence between the number of elements in the parameter array declaration and the number of elements in the actual argument passed in a call.

Coding Guidelines

The information provided by constant expressions appearing within the **[** and **]** of the declaration of a parameter, having an array type, can be of use to static analysis tools. However, in practice because no semantics was associated with such usage in C90, such arrays were rarely declared. It remains to be seen how the semantics given to this usage in C99 will change the frequency of occurrence of parameters having an array type (i.e., will developers use this construct to provide information to translators that might enable them to generate higher-quality code, or to source code analysis tools to enable them to issue better quality diagnostics).

Example

Rationale

```
void fadd(      double a[static restrict 10],
          const double b[static restrict 10])
{
    int i;

    for (i = 0; i < 10; i++) {
        if (a[i] < 0.0)
            return;

        a[i] += b[i];
    }

    return;
}
```

This function definition specifies that the parameters `a` and `b` are restricted pointers. This is information that an optimizer can use, for example, to unroll the loop and reorder the loads and stores of the elements referenced through `a` and `b`.

1600 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

function type adjust to pointer to

Commentary

This is a requirement on the implementation. Occurrences of an object, not declared as a parameter, having a function type are implicitly converted to a pointer type in most contexts.

732 function designator converted to type

Other Languages

Languages that support parameters having some form of function type usually have their own special rules for handling them. A few languages treat function types as first class citizens and they are treated the same as any other type.

Coding Guidelines

While developers might not be aware of this implicit conversion, their interpretation of the behavior for uses of the parameter is likely to match what actually occurs (experience suggests that the lack of detailed knowledge of behavior is not replaced by some misconception that alters developer expectations of behavior).

---

ellipsis supplies no information

If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[123)] 1601

Commentary

That is, no information is supplied by the developer to the translator. The ellipsis notation is intended for use in passing a variable number of argument values (at given positions in the list of arguments) having different types. The origin of support for variable numbers of arguments was the desire to treat functions handling input/output in the same as any other function (i.e., the handling of I/O functions did not depend on special handling by a translator, such as what is needed in Pascal for the `read` and `write` functions).

Prior to the publication of the C Standard there existed a programming technique that relied on making use of information on an implementation's argument passing conventions (invariable on a stack that either grew up or down from the storage location occupied by the last named parameter). Recognizing that developers sometimes need to define functions that were passed variable numbers of arguments the C Committee introduced the ellipsis notation, in function prototypes. The presence of an ellipsis gives notice to a translator that different argument types may be passed in calls to that function. Access to any of these arguments is obtained by encoding information on the expected ordering and type, via calls to library macros, within the body of the function.

C++

The C++ Standard does not make this observation.

Other Languages

The need to pass variable number of arguments, having different types, is a common requirement for languages that specify functions to handle I/O (some languages, e.g., Fortran, handle I/O as part of the language syntax). Many languages make special cases for some I/O functions, those that are commonly required to input or output a number of different values of different types. Having to call a function for each value and the appropriate function used depending on the type of the value is generally thought onerous by language designers.

Coding Guidelines

Many coding guideline documents recommend against the use of ellipsis. The view being taken that use of this notation represents an open-ended charter for uncontrolled argument passing. What are the alternative and how would developers handle the lack of an ellipsis notation? The following are two possibilities:

- *Use file scope objects.* Any number of file scope objects having any available type could be declared to be visible to a function definition and the contexts in which it is called.
- *Use of unions and dummy parameters.* In practice, most functions are passed a small number of optional arguments. A function could be defined to take the maximum number of arguments. In those cases where a call did not need to pass values to all the arguments available to it, a dummy argument could be passed. The number of different argument types is also, usually, small. A union type could be used to represent them.

In both cases the body of the function needs some method of knowing which values to access (as it does when the ellipsis notation is used).

Is the cure worse than the problem? The ellipsis notation has the advantage of not generating new interface issues, which the use of file scope objects is likely to do. The advantage to declaring functions to take the maximum number of arguments (use of union types provides the subset of possible types that argument values

may have) is that information about all possible arguments is known to readers of the function definition. The benefit of the availability of this information is hard to quantify. However, the cost (developer effort required to analyze the arguments in the call, working out which ones are dummy and which unions members are assigned to) is likely to be significant.

Recommending that developers not use the ellipsis notation may solve one perceived problem, but because of the cost of the alternatives does not appear to result in any overall benefit.

While there is existing code that does not use the macros in the **<stdarg.h>** header to access arguments, but makes use of information on stack layout to access arguments, such usage is rarely seen in newly written code. A guideline recommendation dealing with this issue is not considered worthwhile.

1602 The special case of an unnamed parameter of type **void** as the only item in the list specifies that the function has no parameters.

parameter type void

Commentary

The base document had no special syntax for specifying a function declaration that took no parameters. The convention used to specify this case was an empty parameter list. However, an empty parameter list was also used to indicate another convention (which is codified in the standard), a function declaration that provided no information about the arguments it might take (although it might take one or more). Use of the keyword **void** provides a means of explicitly calling out the case of a function taking no parameters (had C90 specified that an empty parameter list denoted a function taking no parameters, almost every existing C program would have become non standards conforming).

1608 function declarator empty list

C90

The C90 Standard was reworded to clarify the intent by the response to DR #157.

Other Languages

Strongly typed languages treat a function declared with no parameters as a function that does not take any arguments, and they sometimes (e.g., Pascal) require the empty parentheses to be omitted from the function call. Other languages vary in their handling of this case.

Example

```
1  typedef void Void;
2
3  extern int f(Void);
4
5  int f(Void)
6  { /* ... */ }
```

1603 If, ~~i~~In a parameter declaration, ~~a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.~~ an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name.

parameter declaration typedef name in parentheses

Commentary

This is a requirement on the implementation. It is an edge case of the language definition. In:

```
1  typedef int T;
2  int f(int *(T));
```

`f` is declared as function returning **int**, taking a single parameter of type pointer to function returning **int** and taking a parameter of type `T`. Without the above rule it could also be interpreted as function returning **int**, taking a single parameter of type pointer to **int**, with redundant parentheses around the identifier `T`.

Wording changes to C90, made by the response to DR #009, were not made to the text of C99 (because the committee thought that forbidding implicit types would eliminate this problem, but an edge case still remained). The response to DR #249 agreed that these changes should have been made and have now been made.

function declarator return type 1592

In the following declaration of `WG14_N852` the identifier `what` is treated as a typedef name and violates the constraint that a function not return a function type.

```
typedef int what;

int WG14_N852(int (what)(int)); /* Constraint violation. */
```

type name empty parentheses 1628

The case of empty parentheses is discussed elsewhere.

C90

The response to DR #009 proposed adding the requirement: "If, in a parameter declaration, an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name."

Other Languages

Most languages do not allow the identifier being declared to be surrounded by parentheses. Neither do they have the C style of declarators.

Coding Guidelines

Developers are unlikely to be familiar with this edge case of the language. However, the types involved (actual and incorrectly deduced) will be sufficiently different that a diagnostic is very likely to be produced for uses of an argument or parameter based on the incorrect type.

If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types. 1604

Commentary

Functions declared with a parameter having an incomplete structure or union type will only be callable after the type is completed. For instance:

```
void f_1(struct T); /* No prior declaration of T visible here, a new type that can never be completed. */
struct T;
void f_2(struct T); /*    Prior declaration of T visible here, refers to existing type. */
/*
 * Cannot define an argument to have type struct T at
 * this point so the function is not yet callable.
 */
struct T {
          int mem;
         };
/*
 * Can now define an object to pass as an argument to f_2.
 */
```

The following declarations of `f` are all compatible with each other (the operand of **`sizeof`** does not need to be evaluated in this context):

```
int f(int n, int a[*]);
int f(int n, int a[sizeof(int [*][*])]);
int f(int n, int a[sizeof(int [n][n+1])]);
```

footnote 109 1465

Use of incomplete types where the size is not needed is discussed elsewhere.

C90

Support for the `[*]` notation is new in C99.

C++

The wording:

> *If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.*[87)] 8.3.5p6

does not contain an exception for the case of a function declaration.

Support for the **[*]** notation is new in C99 and is not specified in the C++ Standard.

Other Languages

Most languages require that structure types appearing in function declarations be complete. A number of languages provide some form of [*] notation to indicate parameters having a variable length array type.

Coding Guidelines

Functions declared with a parameter having an incomplete structure or union type might be regarded as redundant declarations. This issue is discussed elsewhere. 190 redundant code

Example

```
1  void f_1(struct T_1 *p_1);
2
3  struct T_2;
4  void f_2(struct T_2 *p1);
5
6  void f_3(void)
7  {
8  struct T_1 *loc_1 = 0;
9  struct T_2 *loc_2 = 0;
10
11 f_1(loc_1); /* Constraint violation, different structure type. */
12 f_2(loc_2); /* Argument has same structure type as parameter.  */
13 }
```

1605 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

Commentary

The only storage-class specifier that can occur on a parameter declaration is **register**. The interpretation of objects having this storage-class only applies to accesses to them, which can only occur in the body of the function definition. 1593 parameter storage-class

Coding Guidelines

Whether or not function declarations are token for token identical to their definitions is not considered worthwhile addressing in a guideline recommendation.

1606 An identifier list declares only the identifiers of the parameters of the function.

Commentary

In a function declaration such a list provides no information to the translator, but it may provide useful commentary for readers of the source (prior to the availability of function prototypes). In a function definition this identifier list provides information to a translator on the number and names of the parameters.

C++

This form of function declarator is not available in C++.

An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. 1607

Commentary

For a definition of a function, for instance `f`, there is no difference between the forms `f()` and `f(void)`. There is a difference for declarations, which is covered in the following C sentence.

Other Languages

An empty list is the notation commonly used in other languages to specify that a function has no parameters. Some languages also require that the parentheses be omitted.

Coding Guidelines

Differences in the costs and benefits of using either an empty list or requiring the use of the keyword **`void`** are not sufficient to warrant a guideline recommendation dealing with this issue.

function declarator empty list

The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.[124)] 1608

Commentary

This specification differs from that of a function declarator that whose parameter list contains the keyword **`void`**.

parameter 1602 type void

C++

The following applies to both declarations and definitions of functions:

8.3.5p2 *If the `parameter-declaration-clause` is empty, the function takes no arguments.*

A call made within the scope of a function declaration that specifies an empty parameter list, that contains arguments will cause a C++ translator to issue a diagnostic.

Common Implementations

Some translators remember the types of the arguments used in calls to functions declared using an empty list. Inconsistencies between the argument types in different calls being flagged as possibly coding defects.

Coding Guidelines

function declaration use prototype 1810.1

The guideline recommendation specifying the use of function prototypes is discussed elsewhere.

footnote 123

123) The macros defined in the **`<stdarg.h>`** header (7.15) may be used to access arguments that correspond to the ellipsis. 1609

Commentary

These are the `va_*` macros specified in the library section.

footnote 124

124) See "future language directions" (6.11.6). 1610

function compatible types

For two function types to be compatible, both shall specify compatible return types.[125)] 1611

Commentary

compatible type if 631

This is a necessary conditions for two function declarations to be compatible. The second condition is specified in the following C sentence.

C++

compatible type if 631

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

3.5p10

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .

All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type. 8.3.5p3

If one return type is an enumerated type and the another return type is the compatible integer type. C would consider the functions compatible. C++ would not consider the types as agreeing exactly.

Other Languages

Compatibility of function types only becomes an issue when a language's separate translation model allows more than one declaration of a function, or when pointers to functions are supported. In these cases the requirements specified are usually along the lines of those used by C.

Coding Guidelines

The rationale for the guideline recommendations on having a single textual declaration and including the header containing it in the source file that defines the function is to enable translators to check that the two declarations are compatible.

422.1 identifier declared in one file
1818.1 identifier definition shall #include

1612 Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator;

Commentary

This condition applies if both function declarations use prototypes.

C++

A parameter type list is always present in C++, although it may be empty.

Other Languages

Most other languages require that the parameter type lists agree in the number of parameters.

Coding Guidelines

If the guideline recommendation specifying the use of prototypes is followed the parameter type lists will always be present.

1810.1 function declaration use prototype

1613 corresponding parameters shall have compatible types.

Commentary

This requirement applies between two function declarations, not between the declaration of a function and a call to it.

998 function call arguments agree with parameters

C++

All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type. 8.3.5p3

The C++ Standard does not define the concept of compatible type, it requires types to be the same. If one parameter type is an enumerated type and the corresponding parameter type is the corresponding compatible integer type. C would consider the functions to be compatible, but C++ would not consider the types as being the same.

631 compatible type if

Other Languages

Most languages require the parameter types to be compatible.

If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. 1614

Commentary

This C sentence deals with the case of a function prototype (which may or may not be a definition) and an old style function declaration that is not a definition. One possible situations where it can occur is where a function definition has been rewritten using a prototype, but there are still calls made to it from source where an old style declaration is visible. Function prototypes were introduced in C90 (based on the C++ specification). The committee wanted to ensure that developers could gradually introduce prototypes in to existing code. For instance, using prototypes for newly written functions. It was therefore necessary to deal with the case of source code containing so called *old style* and function prototype declarations for the same function.

default argument promotions 1009

Calls where the visible function declaration uses an old style declaration, have their arguments promoted using the default argument promotions. The types of the promoted arguments are required to be compatible with the parameter types in the function definition (which uses a prototype). This requirement on the parameter types in the function definition ensures that argument/parameter storage layout calculations (made by an implementation) are consistent.

The ellipsis terminator is a special case. Some translators are known to handle arguments passed to this parameter differently than when there is a declared type (the unknown nature of the arguments sometimes means that this special processing is a necessity). Given the possibility of this implementation technique the Committee decided not to require the behavior to be defined if the two kinds of declarations were used.

arguments same number as parameters 1010

There can be no check on the number of parameters in the two declarations, since one of them does not have any. This issue is covered elsewhere.

C++

The C++ Standard does not support the C identifier list form of parameters. An empty parameter list is interpreted differently:

8.3.5p2 *If the* `parameter-declaration-clause` *is empty, the function takes no arguments.*

The two function declarations do not then agree:

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .*

A C++ translator is likely to issue a diagnostic if two declarations of the same function do not agree (the object code file is likely to contain function signatures, which are based on the number and type of the parameters in the declarations).

Other Languages

Very few languages (Perl does) have to deal with the issue of developers being able to declare functions using two different sets of syntax rules.

Common Implementations

Implementations are not required to, and very few do, issue diagnostics if these requirements are not met. Whether programs fail to work as expected, if these requirements are not met, often depends on the characteristics of the processor. For those processors that have strict alignment requirements translators

alignment 39

usually assign parameters at least the same alignment as those of the type **int** (ensuring that integer types with less rank are aligned on the storage boundaries of their promoted type). For processors that have more relaxed alignment requirements, or where optimisations are possible for the smaller integer types, parameters having a type whose rank less than that of the type **int** are sometime assigned a different storage location than if they had a type of greater rank. In this case the arguments, which will have been treated as having at least type **int**, will be at different storage locations in the function definitions stack frame.

Coding Guidelines

This case shows that gradually introducing function prototypes into existing source code can cause behavioral differences that did not previously exist. Most of the benefits of function prototype usage come from the checks that translators perform at the point of call. Defining a function using prototype notation and having an old style function declaration visible in a header offers little benefit (unless the function has internal linkage and is visible at all the places it is called). If an existing old style function definition is modified to use a function prototype in its definition, then it is effectively new code and any applicable guideline recommendations apply.

1615 If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier.

Commentary

In this C sentence the two types are a function declaration that uses a prototype and a function definition that uses an old style declaration. In both cases the developer has specified the number and type of two sets of parameters. Both declarations are required to agree in the number of parameters (if the number and type of each parameter agrees there cannot be an ellipsis terminator in the function prototype).

A function defined using an identifier list will be translated on the basis that the arguments, in calls to it, have been promoted according to the default argument promotions. Calls to functions where the visible declaration is a function prototype will be translated on the basis that the definition expects the arguments to be converted at the point of call (to the type of the corresponding parameter). This C sentence describes those cases where the two different ways of handling arguments results in the same behavior. In all other cases the behavior is undefined.

1009 default argument promotions

C++

The C++ Standard does not support the C identifier list form of parameters.

If the `parameter-declaration-clause` *is empty, the function takes no arguments.* 8.3.5p2

The C++ Standard requires that a function declaration always be visible at the point of call (5.2.2p2). Issues involving argument promotion do not occur (at least for constructs supported in C).

```
1  void  f(int, char);
2  void  f(char, int);
3  char a, b;
4  f(a,b);             // illegal: Which function is called?  Both fit
5                      // equally well (equally badly).
```

Common Implementations

Implementations are not required to, and very few do, issue diagnostics if these requirements are not met.

Coding Guidelines

Adding prototype declarations to an existing program may help to detect calls made using arguments that are not compatible with the corresponding functions parameters, but they can also change the behavior of correct

identifier 1818.1 definition shall #include

calls if they are not properly declared. Adhering to the guideline recommendation specifying that the header containing the function prototype declaration be **#include**d in the source file that defines the function is not guaranteed to cause a translator to issue a diagnostic if the above C sentence requirements are not met. The following guideline recommendation addresses this case.

Cg 1615.1

If a program contains a function that is declared using both a prototype and an old style declaration, then the type of each parameter in the prototype shall be compatible with the type of corresponding parameter in the old style declaration after the application of the default argument promotions to those parameter types.

Example

file_1.c

```
int f(p_1)
signed char p_1; /* Expecting argument to have been promoted to int. */
{
return p_1+1;
}
```

file_2.c

```
int f(signed char p_1);

int g(void)
{
return f('0'); /* Argument converted to type signed char. */
}
```

---

parameter qualifier in composite type

(In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.) 1616

Commentary

parameter type 1835 adjusted function 646 composite type compatible type 631 if composite type 642

This specifies the relative ordering of requirements on adjusting types, creating composite types and determining type compatibility. While the composite type of a parameter is always its unqualified type, the wording of the response to DR #040 question 1 explains how composite types are to be treated.

The type of a parameter is independent of the composite type of the function, . . .

DR #040 question 1

In the body of a function the type of a parameter is the type that appears in the function definition, not any composite type. In the following example DR_040_a and DR_040_b have the same composite types, but the parameter types are not the same in the bodies of their respective function definitions.

```
void DR_040_a(const int c_p);
void DR_040_a(      int   p)
{
p=1;                        /* Not a constraint violation. */
}

void DR_040_b(      int   p);
void DR_040_b(const int c_p)
{
c_p=1;                      /*     A constraint violation. */
}
```

Calls to functions will make use of information contained in the composite type. The fact that any parameter types, in the composite type, will be unqualified is not significant because it is the unqualified parameter type that is used when processing the corresponding arguments.

649 prior declaration visible

999 argument type may be assigned

C90

The C90 wording:

> *(For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 6.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)*

was changed by the response to DR #013 question 1 (also see DR #017q15 and DR #040q1).

C++

The C++ Standard does not define the term *composite type*. Neither does it define the concept of compatible type, it requires types to be the same.

642 composite type

631 compatible type if

The C++ Standard transforms the parameters' types and then:

> *If a* `storage-class-specifier` *modifies a parameter type, the specifier is deleted. [Example:* `register char*` *becomes* `char*` *—end example] Such* `storage-class-specifiers` *affect only the definition of the parameter within the body of the function; they do not affect the function type. The resulting list of transformed parameter types is the function's parameter type list.*

8.3.5p3

It is this parameter type list that is used to check whether two declarations are the same.

Coding Guidelines

Adhering to the guideline recommendation specifying the use of function prototypes does not guarantee that the composite type will always contain the same parameter type information as in the original declarations.

1810.1 function declaration use prototype

It is possible that while reading the source of a function definition a developer will make use of information, that exists in their memory, that is based on the functions declaration in a header, rather than the declaration at the start of the definition. The consequences of this usage, in those cases where the parameter types differ in qualification, do not appear to be sufficiently costly (in unintended behavior occurring) to warrant a guideline recommendation.

Example

```
1  void f_1(int  p_a[3]);
2  void f_1(int *p_a   );      /* Compatible with previous f_1 */
```

1617 EXAMPLE 1 The declaration

EXAMPLE function returning pointer to

```
int f(void), *fip(), (*pfi)();
```

declares a function **f** with no parameters returning an **int**, a function **fip** with no parameter specification returning a pointer to an **int**, and a pointer **pfi** to a function with no parameter specification returning an **int**. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the pointer result to yield an **int**. In the declarator **(*pfi)()**, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an **int**.

If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions **f** and **fip** have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer **pfi** has block scope and no linkage.

Commentary

EXAMPLE 1587 array of pointers

The algorithm for reading declarations involving both function and pointer types follows a right then left rule similar to that used for reading declarations involving array and pointer types. However, it is not possible to declare a function of functions, so only one function type on the right is *consumed*.

C++

Function declared with an empty parameter type list are considered to take no arguments in C++.

EXAMPLE 2 The declaration 1618

```
int (*apfi[3])(int *x, int *y);
```

declares an array **apfi** of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers **x** and **y** are declared for descriptive purposes only and go out of scope at the end of the declaration of **apfi**.

Commentary

array element 1567 not function type

The parentheses are necessary because `int *apfi[3](int *x, int *y);` declares `apfi` to be an array of three functions returning pointer to **int** (which is a constraint violation).

EXAMPLE 3 The declaration 1619

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function **fpfi** that returns a pointer to a function returning an **int**. The function **fpfi** has two parameters: a pointer to a function returning an **int** (with one parameter of type **long int**), and an **int**. The pointer returned by **fpfi** points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

Commentary

The declaration

```
1  int (* (*fpfpfi(int (*)(long), int))(int, ...))(void);
```

declares a function `fpfpfi` that returns a pointer to a function returning a pointer to a function returning an **int** involves parenthesizing part of the existing declaration and adding information on the parameters (in this case `(void)`).

---

footnote 125

125) If both function types are "old style", parameter types are not compared. 1620

Commentary

For *old style* function types there is no parameter information to compare (technically there is information available in one case, when one is a definition; however, the standard considers this information to be local to the function body).

C++

The C++ Standard does not support *old style* function types.

Other Languages

Some languages do not specify whether declarations of the same function should be compared for compatibility.

EXAMPLE 4 The following prototype has a variably modified parameter. 1621

```
void addscalar(int n, int m,
       double a[n][n*m+300], double x);
```

```
int main()
{
        double b[4][308];
        addscalar(4, 2, b, 2.17);
        return 0;
}

void addscalar(int n, int m,
        double a[n][n*m+300], double x)
{
        for (int i = 0; i < n; i++)
                for (int j = 0, k = n*m+300; j < k; j++)
                        // a is a pointer to a VLA with n*m+300 elements
                        a[i][j] += x;
}
```

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

The expression n*m+300 occurs in a number of places in the source. Replacing this expression with a symbolic name will reduce the probability of future changes to one use of this expression not being reflected in other uses.

1622 EXAMPLE 5 The following are all compatible function prototype declarators.

EXAMPLE compatible function prototypes

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

(Note that the last declaration also specifies that the argument corresponding to **a** in any call to **f** must be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Commentary

The conversion of parameters having array type to pointer type allows the **restrict** type qualifier to occur in this context.

729 array converted to pointer

1623 **Forward references:** function definitions (6.9.1), type names (6.7.6).

6.7.6 Type names

1624

abstract declarator syntax

type-name:
        specifier-qualifier-list abstract-declarator$_{opt}$
abstract-declarator:
        pointer
        pointer$_{opt}$ *direct-abstract-declarator*
direct-abstract-declarator:

```
( abstract-declarator )
direct-abstract-declarator_opt [ assignment-expression_opt ]
direct-abstract-declarator_opt [ type-qualifier-list_opt assignment-expression_opt ]

direct-abstract-declarator_opt [ static type-qualifier-list_opt assignment-expression ]
direct-abstract-declarator_opt [ type-qualifier-list static assignment-expression ]
direct-abstract-declarator_opt [ * ]
direct-abstract-declarator_opt ( parameter-type-list_opt )
```

Commentary

An abstract declarator specifies a type without defining an associated identifier. The term *type-name* is slightly misleading since there is no name, the type is anonymous. The *direct-declarator* productions:

declarator 1547 syntax

```
direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
direct-declarator [ static type-qualifier-list_opt assignment-expression ]
direct-declarator [ type-qualifier-list static assignment-expression ]
```

are not supported for abstract declarators (some committee members say this was intended, others that it was an accidental omission).

The wording was changed by the response to DR #289 and makes the syntax consistent with that for *direct-declarator*.

declarator 1547 syntax

C90

Support for the form:

```
direct-abstract-declarator_opt [ * ]
```

is new in C99. In the form:

```
direct-abstract-declarator_opt [ assignment-expression^opt ]
```

C90 only permitted *constant-expression*$_{opt}$ to appear between **[** and **]**.

C++

The C++ Standard supports the C90 forms. It also includes the additional form (8.1p1):

```
direct-abstract-declarator_opt ( parameter-declaration-clause )
cv-qualifier-seq_opt exception-specification_opt
```

Other Languages

A few languages (e.g., Algol 68) have a concept similar to that of abstract declarator (i.e., an unnamed type that can appear in certain contexts, such as casts).

Coding Guidelines

The term *type* is applied generically to all type declarations, whether they declare identifiers or not. Developers do not appear to make a distinction between declarators and abstract declarators.

More cognitive effort is needed to comprehend an abstract declarator than a declarator because of the additional task of locating the position in the token sequence where the identifier would have been, had it not

been omitted. Whether there is sufficient benefit in providing an identifier (taking into account the costs of providing it in the first place) to make a guideline recommendation worthwhile is a complex question that your author does not yet feel capable of answering.

Semantics

1625 In several contexts, it is necessary to specify a type.

Commentary

These contexts are: a compound literal, the type in a cast operation, the operand of **sizeof**, and parameter types in a function prototype declaration that omits the identifiers.

1626 This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[126)]

Commentary

This defines the term *type name* (and saying in words what is specified in the syntax).

C++

Restrictions on the use of type names in C++ are discussed elsewhere.

1118 sizeof constraints
1134 cast scalar or void type
1592 function declarator return type

Coding Guidelines

A *type-name* is syntactically a declaration and it is possible to declare identifiers using it. For instance:

```
1  void f_1(int p)
2  {
3  p=(enum {apple, orange, pair})sizeof(enum {brazil, cashu, almond});
4  }
5
6  struct T {int mem;} f_2(void)
7  {
8  struct T loc;
9  /* ... */
10 return loc;
11 }
```

Such uses are not common and might come as a surprise to some developers. The cost incurred by this usage is that readers of the source may have to spend additional time searching for the identifiers declared, because they do not appear in an expected location. There is no obvious worthwhile benefit (although there is a C++ compatibility benefit) in a guideline recommendation against this usage.

1627 EXAMPLE The constructions

EXAMPLE abstract declarators

```
(a)     int
(b)     int *
(c)     int *[3]
(d)     int (*)[3]
(e)     int (*)[*]
(f)     int *()
(g)     int (*)(void)
(h)     int (*const [])(unsigned int, ...)
```

name respectively the types (a) **int**, (b) pointer to **int**, (c) array of three pointers to **int**, (d) pointer to an array of three **int**s, (e) pointer to a variable length array of an unspecified number of **int**s, (f) function with no parameter specification returning a pointer to **int**, (g) pointer to function with no parameters returning an **int**, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type **unsigned int** and an unspecified number of other parameters, returning an **int**.

Commentary

The following is one algorithm for locating where the omitted identifier occurs in an abstract declarator. Starting on the left and working right:

1. skip all keywords, identifiers, and any matched pairs of braces along with their contents (the latter are **struct/union/enum** declarations), then
2. skip all open parentheses, asterisks, and type qualifiers. The first unskipped token provides the context that enables the location of the omitted identifier to be deduced:
   - A **[** token is the start of an array specification that appears immediately after the omitted identifier.
   - A *type-specifier* is the start of the *declaration-specifier* of the first parameter of a parameter list. The omitted identifier occurs immediately before the last skipped open parenthesis.
   - A **)** token immediately after a **(** token is an empty parameter list. The omitted identifier occurs immediately before the last skipped open parenthesis.
   - A **)** token that is not immediately after a **(** token is the end of a parenthesized abstract declarator with no array or function specification. The omitted identifier occurs immediately before this **)** token.

C90

Support for variably length arrays is new in C99.

C++

Support for variably length arrays is new in C99 and is not specified in the C++ Standard.

---

footnote 126 type name empty parentheses

126) As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification", rather than redundant parentheses around the omitted identifier. 1628

Commentary

That is in the following declaration of `f`:

```
1  typedef char x;
2  void f(int (x), /* Function returning int having a single int parameter. */
3         int ()); /* Function returning int with no parameter information specified. */
4
5  void g(int (y)  /* Parameter having type int and name y. */
6        );
```

parameter 1603 declaration typedef name in parentheses

The behavior of parentheses around an identifier is also described elsewhere.

Other Languages

This notation is used by many languages (many of which don't allow parentheses around identifiers).

Coding Guidelines

function declaration use prototype 1810.1

The guideline recommendation dealing with the use of prototypes in function declarators is applicable here.

6.7.7 Type definitions

typedef name syntax

1629

typedef-name:

identifier

Commentary

A typedef name exists in the same name space as ordinary identifiers. The information that differentiates an identifier as a *typedef-name*, from other kinds of identifiers is the visibility, or not, of a typedef definition of that identifier. For instance, given the declarations:

444 name space ordinary identifiers

```
typedef int type_ident;
type_ident(D_1);           /* Function call or declaration of D_1?  */
type_ident * D_2;          /* Multiplication or declaration of D_2? */
```

it is not possible to decide, using syntax only, whether `type_ident(D_1);` is a function call or a declaration of `D_1` using redundant parentheses. In the declaration of `D_2` a parser does not know this is a declaration, rather than a syntax violation, until it has seen the sixth token after the `type_ident` at the start of the line (i.e., more than one token lookahead is required).

There are some contexts where the status of an identifier as a *typedef-name* can be deduced. For instance, the token sequence `; x y;` is either a declaration of `y` to have the type denoted by `x`, or it is a violation of syntax (because a definition of `x` as a typedef name is not visible).

Other Languages

Languages invariably use ordinary identifiers to indicate both objects and their equivalent (if supported) of typedef names.

Common Implementations

The syntax of most languages is such that it is possible to parse their source without the need for a symbol table holding information on prior declarations. It is also usually possible to parse them by looking ahead a single token in the input stream (tools such as `bison` support such language grammars).

The syntax of C declarations and the status of *typedef-name* as an identifier token creates a number of implementation difficulties. The parser either needs access to a symbol table (so that it knows which identifiers are defined as *typedef-name*s), or it needs to look ahead more than one token and be able to handle more than one parse of some token sequences. Most implementations use the symbol table approach (which in practice is more complicated than simply accessing a symbol table; it is also necessary to set or reset a flag based on the current syntactic context, because an identifier should only be looked up, to find out if it is currently defined as a *typedef-name*, in a subset of the contexts in which an identifier can occur).

Coding Guidelines

It is common developer practice to use the term *type name* (as well as the term *typedef name*) to refer to the identifier defined by a typedef declaration. There is no obvious benefit in attempting to change this common developer usage. The issue of naming conventions for typedef names is discussed elsewhere.

792 typedef naming conventions

The higher-level source code design issues associated with the use of typedef names are discussed below. Given some of the source reading techniques used by developers it is possible that a typedef name appearing in a declaration will be treated as one of the identifiers being declared. This issue is discussed elsewhere. This coding guideline subsection discusses the lower-level issues, such as processing of the visible source by readers and naming conventions.

770 reading kinds of 1348 declaration syntax

The only time the visible form of declarations is likely to contain two identifiers adjacent to each other (separated only by white space) is when a typedef name is used (such adjacency can also occur through the use of macro names, but is rare). The two common cases are for the visible source lines, containing a declaration, to either start with a keyword (e.g., **`int`** or **`struct`**) or an identifier that is a member of an identifier list (e.g., a list of identifiers being declared).

Some of the issues involved in deciding whether to use a typedef name of a tag name, for structure and union types, is discussed elsewhere. Using a typedef name provides a number of possible benefits, including the following:

792 tag naming conventions

- Being able to change the type used in more than declaration by making a change to one declaration (the typedef name). In practice this cost saving is usually only a significant factor when the type category

553 type category

is not changed (e.g., an integer type is changed to an integer type, or a structure type is changed to another structure type). In this case the use of objects, declared using these typedef name, as operands in expressions does not need to be modified (changing, for instance, an array type to a structure type is likely to require changes to the use of the object in expressions).

- The spelling of the type name may providing readers with semantic information about the type that would not be available if the sequence of tokens denoting the type had appeared in the source. The issue of providing semantic information via identifier spellings is discussed elsewhere.

identifier 792
semantic associations

- The use of typedef names is sometimes recommended in coding guideline documents because it offers a mechanism for hiding type information. However, readers are likely to be able to deduce this (from looking at uses of an object), and are also likely to need to know an objects type category (which is probably the only significant information that type abstraction is intended to hide).

type category 553

Usage

A study by Neamtiu, Foster, and Hicks[998] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 16% of releases one or more existing typedef names had the type they defined changed.[997]

Table 1629.1: Occurrences of types defined in a **typedef** definition (as a percentage of all types appearing in **typedef** definition). Based on the translated form of this book's benchmark programs.

Type	Occurrences	Type	Occurrences
struct	58.00	**unsigned long**	1.47
enum	9.50	**int** *()	1.46
other-types	8.86	**enum** *()	1.46
struct *	6.97	**union**	1.38
unsigned int	2.68	**long**	1.29
int	2.46	**void** *()	1.18
unsigned char	2.21	**unsigned short**	1.07

Constraints

If a typedef name specifies a variably modified type then it shall have block scope. 1630

Commentary

Allowing a typedef name to occur at file scope appears to be useful; the identifier name provides a method of denoting the same type in declarations in different functions, or translation units. However, the expression denoting the number of elements in the array has to be evaluated during program execution. The committee decided that this evaluation would occur when the type declaration was encountered during program execution. This decision effectively prevents any interpretation being given tor such declarations at file scope.

array size 1632
evaluated when declaration reached

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and not specified in the C++ Standard.

Coding Guidelines

The extent to which more than one instance of a variably modified type using the same size expression and element type will need to be defined in different functions is not known. A macro definition provides one mechanism for ensuring the types are the same. Variably modified types are new in C99 and experience with their use is needed before any guideline recommendations can be considered.

Semantics

1631 In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.5.

Commentary

The association between **typedef** and storage class is a syntactic one (they share the same declarator forms), not a semantic one.

Other Languages

Many languages that support developer defined type names define a syntax that is specific to that usage (which may simply involve the use of different keyword, for instance Pascal requires that type definitions be introduced using the keyword **type**).

Coding Guidelines

The issues involved in declarations that declare more than one identifier are discussed elsewhere.

1348 declaration visual layout

Example

```
1  extern  int i, j[3];
2  typedef int I, J[3];
```

1632 Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution.

array size evaluated when declaration reached

Commentary

Using a **typedef** to declare a variable length array object (see §6.7.5.2) could have two possible meanings. Either the size could be eagerly computed when the **typedef** is declared, or the size could be lazily computed when the object is declared. For example Rationale

```
{
    typedef int VLA[n];
    n++;
    VLA object;

    // ...
}
```

The question arises whether `n` should be evaluated at the time the type definition itself is encountered or each time the type definition is used for some object declaration. The Committee decided that if the evaluation were to take place each time the typedef name is used, then a single type definition could yield variable length array types involving many different dimension sizes. This possibility seemed to violate the spirit of type definitions. The decision was made to force evaluation of the expression at the time the type definition itself is encountered.

In other words, a typedef declaration does not act like a macro definition (i.e., with the expression evaluated for every invocation), but like an initialization (i.e., the expression is evaluated once).

C90

Support for variable length array declarators is new in C99.

C++

Support for variable length array declarators is new in C99 and is not specified in the C++ Standard.

Other Languages

Other languages either follow the C behavior (e.g., Algol 68), or evaluated on each instance of a typedef name usage (e.g., Ada).

Coding Guidelines

While support for this construct is new in C99 and at the time of this writing insufficient experience with its use is available to know whether any guideline recommendation is worthwhile, variable length array declarations can generate side effects, a known problem area. The guideline recommendation applicable to side effects in declarations is discussed elsewhere.

full declarator 187.2
all orderings give same type

Example

In the following the objects `q` and `r` will contain the same number of elements as the object `p`.

```
1  void f(int n)
2  {
3  START_AGAIN: ;
4
5  typedef int A[n];
6
7  A p;
8  n++;
9  A q;
10
11    {
12    int n = 99;
13    A r; /* Uses the object n visible when the typedef was defined. */
14    }
15
16 if ((n % 4) != 0)
17    goto START_AGAIN;
18
19 if ((n % 5) != 0)
20    f(n+2)
21 }
```

---

typedef is synonym

A **`typedef`** declaration does not introduce a new type, only a synonym for the type so specified. 1633

Commentary

Typedef names provide a mechanism for easily modifying (by changing a single definition) the types of a set of objects (those declared using a given typedef name) declared within the source code. The fact that a **`typedef`** only introduces a synonym for a type, not a new type, is one of the reasons C is considered to be a typed language but not a strongly typed language.

It is not possible to introduce a new name for a tag. For instance:

```
1  typedef oldtype newtype;                /* Supported usage.  */
2  typedef struct oldtype struct newtype;  /* Syntax violation. */
```

One difference between a typedef name and a tag is that the former may include type qualifier information. For instance, in:

```
1  typedef const struct T {
2                           int mem;
3                          } cT;
```

the typedef name cT is **`const`** qualified, while the tag T is not.

Other Languages

Some languages regard all typedef declarations as introducing new types. A few languages have different kinds of typedef declarations, one introducing a new type and the other introducing a synonym.

Common Implementations

Some static analysis tools support an option that allows typedef names to be treated as new (i.e., different) types (which can result in diagnostics being issued when mixing operands having types).

Coding Guidelines

Experienced users of strongly typed languages are aware of the advantages of having a typedef that creates a new type (rather than a synonym). They enable the translator to aid the developer (by detecting mismatches and issuing diagnostics) in ensuring that objects are not referenced in inappropriate contexts. Your author is not aware of any published studies that have investigated the costs and benefits of strong typing (there have been such studies comparing so called *strongly typed* languages against C, but it is not possible to separate out the effects of typing from other language features). Also there does not appear to have been any work that has attempted to introduce strong typing into C in a commercial environment.

Your authors experience (based on teaching Pascal and analyzing source code written in it) is that it takes time and practice for developers to learn how to use strong type names effectively. While the concepts behind individual type names may be quickly learned and applied, handling the design decisions behind the interaction between different type names requires a lot of experience. Even in languages such as Pascal and Ada, where implementations enforced strong typing, developers still required several years of experience to attain some degree of proficiency.

Given C's permissive type checking (e.g., translators are not required to perform much type checking on the standard integer types), only a subset of the possible type differences are required to generate a diagnostic to be generated. Given the lack of translator support for strong type checking and the amount of practice needed to become proficient in its use, there is no cost/benefit in recommending the use of typedef names for type checking purposes. The cost/benefit of using typedef names for other purposes, such as enabling the types of a set of objects to be changed by a single modification to the source, may be worthwhile.

493 standard integer types

Measurements of the translated form of this book's benchmark programs show that typedef names occur much more frequently than the tag names (by a factor of 1.7:1; although this ratio is switched in some programs, e.g., tag names outnumber typedef names by 1.5:1 in the Linux kernel). Why is this (and should the use of typedef names be recommended)? The following are some of the possible reasons:

- Developers new to C imitate what they see others doing and what they read in books.
- The presence of the **`struct`**/**`union`**/**`enum`** keyword is considered to be useful information, highlighting the kind of type that is being used (structure types in particular seem to be regarded as very different animals than other types). While this rationale goes against the design principle of hiding representation details, experience shows that uses of structure types are rarely changed to non-structure types.
- The usage is driven by the least effort principle being applied by a developer at the point where the structure type is defined. While the effort needed in subsequent references to the type may be less, had a typedef name had been used, the cost of subsequent uses is not included in the type definition decision process.

1634 That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in **`T`** (known as *T*), and the identifier in **`D`** has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of **`D`**.

Commentary

Declarations of the form `type_ident D;` is the only situation where two identifier tokens are adjacent in the preprocessed source.

C++

This example and its associated definition of terms is not given in the C++ Standard.

A typedef name shares the same name space as other identifiers declared in ordinary declarators. 1635

Commentary

name space 444 ordinary identifiers

This issue is discussed elsewhere.

EXAMPLE 1 After 1636

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of **`distance`** is **`int`**, that of **`metricp`** is "pointer to function with no parameter specification returning **`int`**", and that of **`x`** and **`z`** is the specified structure; **`zp`** is a pointer to such a structure. The object **`distance`** has a type compatible with any other **`int`** object.

Coding Guidelines

typedef 792 naming conventions

The use of uppercase in typedef names is discussed elsewhere.

EXAMPLE 2 After the declarations 1637

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **`t1`** and the type pointed to by **`tp1`** are compatible. Type **`t1`** is also compatible with type **`struct s1`**, but not compatible with the types **`struct s2`**, **`t2`**, the type pointed to by **`tp2`**, or **`int`**.

Coding Guidelines

member 792 naming conventions

The issue of different structure types declaring members having the same identifier spelling is discussed elsewhere.

EXAMPLE 3 The following obscure constructions 1638

```
typedef signed int t;
typedef int plain;
struct tag {
        unsigned t:4;
        const t:5;
        plain r:5;
};
```

declare a typedef name **`t`** with type **`signed int`**, a typedef name **`plain`** with type **`int`**, and a structure with three bit-field members, one named **`t`** that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it could be accessed) would contain values in either the range [-15, +15] or [-16, +15], and one named **`r`** that contains values in one of the ranges [0, 31], [-15, +15], or [-16, +15]. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **`unsigned`** is a type specifier (which forces **`t`** to be the name of a structure member), while **`const`** is a type qualifier (which modifies **`t`** which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function **f** is declared with type "function returning **signed int** with one unnamed parameter with type pointer to function returning **signed int** with one unnamed parameter with type **signed int**", and an identifier **t** with type **long int**.

Coding Guidelines

The guideline recommendation dealing with the reuse of identifiers is applicable here.

792.3 identifier reusing names

1639 EXAMPLE 4

On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **signal** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Commentary

The algorithm for locating the omitted identifier in abstract declarators can be used to locate the identifier declared by the declarator in complex declarations.

1627 EXAMPLE abstract declarators

1640 EXAMPLE 5 If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
        typedef int B[n];   // B is n ints, n evaluated now
        n += 1;
        B a;                // a is n ints, n without += 1
        int b[n];           // a and b are different sizes
        for (int i = 1; i < n; i++)
                a[i-1] = b[i];
}
```

Other Languages

Some languages (e.g., Ada) support parameterized typedefs that allow information, such as array size, to be explicitly specified when the type is instantiated (i.e., when an object is declared using it).

Coding Guidelines

At the time of this writing there is insufficient experience available with how variable length array types are used to know whether a guideline recommendation dealing with modifications to the values of objects used in the declaration of typedef names, such as n in the above example, is worthwhile.

6.7.8 Initialization

---

initialization syntax

1641

initializer:
    assignment-expression
    { *initializer-list* **}**
    { *initializer-list* **,** **}**
initializer-list:
    $designation_{opt}$ *initializer*
    initializer-list **,** $designation_{opt}$ *initializer*

```
designation:
                designator-list =
designator-list:

                designator
                designator-list designator
designator:
                [ constant-expression ]
                . identifier
```

Commentary

The term *designated initializer* is sometimes used in discussions by members of the committee. However, the C Standard does not use or define this term, it uses the term *designation initializer*.

Rationale

A new feature of C99: Designated initializers provide a mechanism for initializing sparse arrays, a practice common in numerical programming. They add useful functionality that already exists in Fortran so that programmers migrating to C need not suffer the loss of a program-text-saving notational feature.

This feature also allows initialization of sparse structures, common in systems programming, and allows initialization of unions via any member, regardless of whether or not it is the first member.

C90

Support for designators in initializers is new in C99.

C++

Support for designators in initializers is new in C99 and is not specified in the C++ Standard.

Other Languages

Ada supports the initialization of multiple objects with a single value.

```
1  Total_val,
2  Total_average : INTEGER : = 0;
```

Ada (and Extended Pascal) does not require the name of the member to be prefixed with the `.` token and uses the token `=>` (Extended Pascal uses `:`), rather than `=`.

Extended Pascal supports the specification of default initial values in the definition of a type (that are then applied to objects defined to have that type).

The Fortran **DATA** statement is executed once, prior to program startup, and allows multiple objects to be initialized (sufficient values are used to initialize each object and it is the authors responsibility for ensuring that each value is used to initialize the intended object):

```
1       CHARACTER*5 NAME
2       INTEGER I, J, K
3       DATA NAME, I, J, K / 'DEREK', 1, 2, 3/
```

Ada, Extended Pascal, and Fortran (since the 1966 standard) all provide a method of specifying a range of array elements that are to be initialized with some value.

Algol 68 uses the token `=` to specify that the declared identifier is a constant, and `:=` to specify that declared identifier is a variable with an initial value. For instance:

```
INT c = 5;       # constant #
INT d = c + 1;   # constant #
INT e := d;      # initialized variable #
```

BCPL supports parallel declarations, for instance:

```
LET a, b, c = x, y, z; // declare a, b, and c, and then in some order assign x to a, y to b, and z to c
```

Common Implementations

Some prestandard implementations (e.g., `pcc`) parsed initializers bottom-up, instead of top-down (as required by the standard). A few modern implementations provide an option to specify a bottom-up parse (e.g., the Diab Data C compiler[353] supports the `-Xbottom-up-init` option). For instance, in:

```
struct T { int a, b; };
struct  {
          struct T c[2];
          struct T d[2];
         } x = {
                 {1, 2},
                 {3, 4}
                };
```

the initialization of x is equivalent to:

```
{ { {1, 2}, {0, 0} },
  { {3, 4}, {0, 0} } };
```

An implementation performing a bottom-up parse would treat it as being equivalent to:

```
{ { {1, 2}, {3, 4} },
  { {0, 0}, {0, 0} } };
```

`gcc` supports the use of a range notation in array initializer designators. For instance:

```
int my_array[100] = { [0 ... 99] = 1 };
int my_parts[PART_1 + PART_2] = { [0 ... PART_1-1] = 1,
                                  [PART_1 ... PART_1+PART_2-1] = 2 };
```

Coding Guidelines

Constant values often occur in initializer lists. Such occurrences may be the only instance of the constant value in the source, or the values appearing in the list may have semantic or mathematical associations with each other. In these cases the guideline recommendation that names be given to constants may not have a worthwhile benefit.

Dev 825.3 An integer constant may appear in the list of initializers for an object having an array or structure type.

Dev 842.1 A floating constant may appear in the list of initializers for an object.

What are the costs and benefits of organizing the visible appearance of initializers in different ways (like other layout issues, experience suggests that developers have an established repertoire of layout rules which provide the template for laying out individual initializers, e.g., a type based layout with array elements in columns and nested structure types indented like nested conditionals)?

1707 statement visual layout
1348 declaration visual layout

- *Cost*— the time taken to create the visual layout and to maintain it when new initializers are added, or existing ones removed. Experience suggests that developers tend to read aggregate initializers once (to comprehend what their initial value denotes) and ignore them thereafter. Given this usage pattern, initializers are likely to be a visual distraction most of the time (another cost).
- *Benefit*— the visual layout may reduce the effort needed by readers to comprehend them.

Until more is known about the frequency with which individual initializers are read for comprehension, as opposed to being given a cursory glance (almost treated as distractions) it is not possible to reliably provide cost effective recommendations about how to organize their layout. The following discusses some possibilities.

gestalt principles 770

The gestalt principle of organization suggest that related initializers be visually grouped together. It is not always obvious what information needs to be visually grouped. For instance, for an array of structure type initializer might be grouped by array element or by structure member, or perhaps an application oriented grouping:

```
/* Low visual distraction to rest of source. */
{ {1, 'w'}, {2, 'q'}, {3, 'r'} {4, 'a'} }

/* Grouped by array element. */
{
 {1, 'w'},
 {2, 'q'},
 {3, 'r'},
 {4, 'a'}
}

/* Application may suggest an odd/even array element grouping. */
{
 {1, 'w'},
            {2, 'q'},
 {3, 'r'},
            {4, 'a'}
}

/* Perhaps initializers should be grouped by member. */
{
 {1,     'w'},
 {2,     'q'},
 {3,     'r'},
 {4,     'a'}
}
```

Other considerations on initializer layout include making it easy to check that all required initializers are present and visually exposing patterns in the constants appearing in the initializer (with the intent of reducing the effort needed, by subsequent, to deduce them):

```
int bit_pattern[] = {
                     0001, /* octal constant */
                     0010,
                     0100,
                     1000, /* Context may cause reader treat this as an octal constant. */
                    };
```

Other cognitive factors include subitizing and the Stroop effect.

subitizing

When people are asked to enumerate how many dots, for instance, are visible in a well defined area their response time depends on the number of dots. However, when there are between one and four dots performance varies between 40 ms to 100 ms per dot. With five or more dots performance varies between

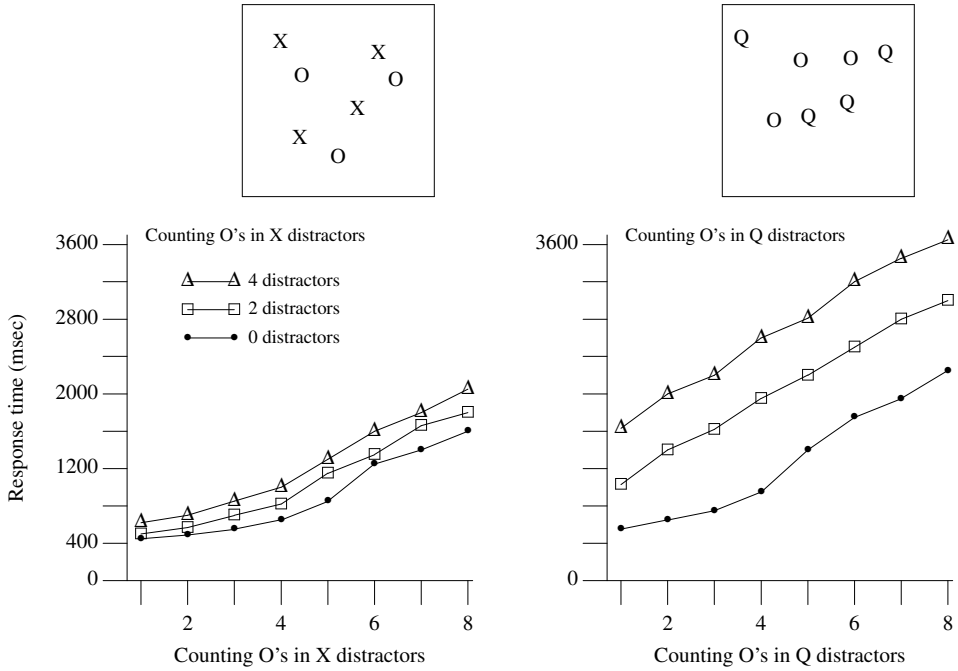


Figure 1641.1: Average time (in milliseconds) taken for subjects to enumerate O's in a background of X or Q distractors. Based on Trick and Pylyshyn.[1365]

250 ms to 350 ms per dot. The faster process used when there are four or fewer dots is called *subitizing* (people effortlessly *see* the number of dots), while the slower process is called *counting*.

Subitizing has been shown[1365] to rely on information available during the preattentive stage of vision. 770 vision preattentive Items that rely on later stages of visual processing (e.g., those requiring spatial attention, such as enumerating the number of squares along a given line) cannot be subitized, they have to be counted. The limit on the maximum number of items that can be subitized is thought to be caused by capacity limits in the preattentive stages of vision.[1366] The extent to which other items, distractors, visible on the display reduce enumeration performance depends on the number of distractors and whether it is possible to discriminate between the visible items during the visual systems preattentive stage. For instance, it is possible to subitize the letter *O* when the distractors are the letter *X*, but not when the distractors are the letter *Q* (see Figure 1641.1).

A study by Stroop[1309] asked subjects to name the color of ink that words were written in. For instance, stroop effect the word *red* was printed in black, blue, and green inks and the word *blue* was printed in black, red, and green inks. The results showed that performance (response time and error rate) suffered substantial interference from the tendency to name the word, rather than its color.

The explanation for what has become known as the *Stroop* effect is based on interference, in the human brain, between the two tasks of reading a word and naming a colour (which are performed in parallel; in general words are read faster, an automatic process in literate adults, than colors can be named). Whether the interference occurs in the output unit, which is serial (one word arriving just before the other and people only being able to say one word at a time), or occurs between the units doing the naming and reading, is still an open question.

Experiments using a number of different kinds of words and form of visual presentation had replicated the effect. For instance, the Stroop effect has been obtained using lists of numbers. Readers might like to try counting the number of characters occurring in each separate row appearing in the margin.

3 3 3 3
a a a a a a
8 8 8
z z
1 1
t t t t
6 6 6 6 6

The effort of counting the digit sequences is likely to have been greater and more error prone than for the letter sequences.

Studies[1064] have found that when subjects are asked to enumerate visually presented digits, the amount of Stroop-like interference depends on the arithmetic difference between the magnitude of the digits used and the number of those digits displayed. Thus a short, for instance, list of large numbers is read more quickly and with fewer errors than a short list of small numbers. Alternatively a long list of small numbers (much smaller than the length of the list) is read more quickly and with fewer errors than a long list of numbers where the number has a similar magnitude to the length of the list.

Initializers often contain lists of similar numbers. The extent to which initializer layout interacts with readers using subitizing/counting and the Stroop effect is not known.

Example

If the `wchar_t` type is different from type **char**, then:

```
#include <stddef.h>

char str1[] = L"abc";               /* Constraint violation, if (wchar_t != char) */
char str2[] = {L'a', L'b', L'c'};   /* OK    */

wchar_t wstr1[] = "abc";            /* Constraint violation, if (wchar_t != char) */
wchar_t wstr2[] = {'a', 'b', 'c'};  /* OK    */
```

Table 1641.1: Occurrence of object types, in block scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table 1348.2).

Type	%	Type	%
struct *	39.5	**long**	2.6
int	22.6	**char**	2.5
other-types	9.1	**unsigned short**	2.4
unsigned int	4.5	**unsigned char**	1.5
union *	4.3	**unsigned char** *	1.4
char *	4.0	**unsigned int** *	1.2
unsigned long	3.4	**enum**	1.1

Table 1641.2: Occurrence of object types with internal linkage, at file scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table 1348.4).

Type	%	Type	%
const char []	22.5	**char** *	2.2
const struct	14.7	**int** []	2.1
int	11.1	**char** []	2.0
struct	10.4	**unsigned char** []	1.7
other-types	10.4	**void** *()	1.3
struct []	8.3	**(char** * **)** []	1.3
struct *	2.9	**int** *()	1.2
(const char * **const)** []	2.9	**const unsigned char** []	1.2
unsigned short []	2.5	**const short** []	1.2

Constraints

initializer value not contained in object

No initializer shall attempt to provide a value for an object not contained within the entity being initialized. 1642

Commentary

This constraint can apply to any initializer where a designator is not used and more values are given than are available to be initialized in the type. It can also apply to an array being initialized with a designator

that specifies an element not contained within the array type (structure member designators are covered by another constraint).

1648 designator
. identifier

C90

There shall be no more initializers in an initializer list than there are objects to be initialized.

Support for designators in initializers is new in C99 and a generalization of the wording is necessary to cover the case of a name being used that is not a member of the structure or union type, or an array index that does not lie within the bounds of the object array type.

C++

The C++ Standard wording has the same form as C90, because it does not support designators in initializers.

An `initializer-list` is ill-formed if the number of initializers exceeds the number of members or elements to initialize. 8.5.1p6

1643 The type of the entity to be initialized shall be an array of unknown size or an object type that is not a variable length array type.

Commentary

An array of unknown size is an incomplete type and therefore not an object type.

546 array
unknown size
475 object types

There is no mechanism for specifying a repeat factor for initializers in C, the number of elements is known at translation time. By the nature of their intended usage the number of elements in an object having a variable length array type is not known at translation time and is likely to vary between different instances of type instantiation during program executions. Providing support for initializations would involve specifying many different combinations of events, a degree of complexity that is probably not worth the cost. It is not possible to specify a single initial value, in the declaration of an object having a variable length array type, as a means of implicitly causing all other elements to be initialized to zero.

1682 initializer
fewer in list than members

C90

Support for variable length array types is new in C99.

1644 All the expressions in an initializer for an object that has static storage duration shall be constant expressions or string literals.

initializer
static storage duration object

Commentary

This requirement ensures that the initial value of objects is known at translation time. This simplifies program startup and avoids the complications involved in deducing dependencies between initialization values (needed to define an order of static initialization, on program startup, that gives consistent behavior).

C90

All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

C99 has relaxed the requirement that aggregate or union types always be initialized with constant expressions.

Support for string literals in this context was added by the response to DR #150.

C++

8.5p2 *Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving literals and previously declared variables and functions.*

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

C++ translators that have an *operate in C mode* option have been known to fail to issue a diagnostic for initializers that would not be acceptable to conforming C translators.

Other Languages

Not all languages require the values of initializers to be known at translation time.

---

identifier linkage at block scope

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier. 1645

Commentary

Existing code, prior to C90, contained declarations of identifiers with external linkage (but without initializers) in block scope and the C committee sanctioned its continued use. Providing an initializer for an object having external or internal linkage, is one method of specifying that it denotes the definition of that object. However, support for such usage has no obvious benefit in block scope.

object 1354 reserve storage

no linkage 435 block scope object

Block scope definitions that include the **`static`** storage-class specifier have no linkage and may contain an initializer.

C++

The C++ Standard does not specify any equivalent constraint.

Coding Guidelines

If the guideline recommendation dealing with declaring identifiers with external or internal linkage at file scope is followed this situation can never occur.

identifier 422.1 declared in one file

---

designator constant-expression

If a designator has the form 1646

```
[ constant-expression ]
```

then the current object (defined below) shall have array type and the expression shall be an integer constant expression.

Commentary

While it is possible for the initialization value to be a nonconstant, the designator of the element being initialized must be constant. Requiring support for nonconstant designators would have introduced a number of complexities. For instance, in the following example:

```
1  int n;
2  /* ... */
3  int vec_1[]   = { [n] = 2 };
4  int vec_2[10] = { [n] = 2 };
```

the declaration of `vec_1` is essentially a new way of specifying a VLA, while in the initializer for `vec_2` a translator cannot enforce the constraint that initializers only provide values for objects contained within the entity being initialized until program execution.

initializer 1642 value not contained in object

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

Some of the coding guideline issues involved in using designators are discussed elsewhere.

1670 initialization using a designator

1647 If the array is of unknown size, any nonnegative value is valid.

Commentary

The number of elements in an array declarator that does not specify a size is deduced from its initializer. Hence, any nonnegative value is guaranteed to be permitted by the standard. However, an implementation may fail to translate a source file containing an object whose size exceeds the minimum required limits.

1683 array of unknown size initialized
294 limit minimum object size

Coding Guidelines

There is not sufficient experience available with the use of designators to know if any particular nonnegative value should be considered suspicious (e.g., very large values, or gaps in a consecutive range).

1648 If a designator has the form

designator . identifier

```
. identifier
```

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

Commentary

This requirement mimics that specified for the `.` operator.

1029 operator . first operand shall

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

The issue of two or more designators specifying a value for the same member of a subobject is discussed elsewhere.

1676 initialization in list order

Semantics

1649 An initializer specifies the initial value stored in an object.

initializer initial value

Commentary

Here the term *initial* refers to the behavior from an executing programs perspective. When initialization occurs depends on the storage duration of the object being initialized. Objects with static storage duration are initialized on program startup, while objects with automatic storage duration are initialized when the objects declaration is encountered during program execution (although their lifetime starts when the block that contains the declaration is entered), and objects with allocated storage duration can be given an initial value to zero by calling the `calloc` library function.

151 static storage duration initialized before startup
1711 object initializer evaluated when
451 lifetime of object

Common Implementations

The machine code generated to initialize scalar objects with automatic storage duration is usually the same as that used to assign a value in an expression statement.

It is much easier to generate efficient machine code for aggregate objects, with automatic storage duration, when an initializer is used compared to when the developer has used a sequence of assignment statements (the translator does not need to perform any analysis to deduce that the values being assigned are all associated with the same object). Using compound literals is likely to result in translators allocating additional storage for the unnamed object (unless it can be deduced that such storage is unnecessary).

1058 compound literal unnamed object

Coding Guidelines

Some coding guideline documents recommend that an initializer always be used to store an initial value in an object. The rationale being that providing an initial value in the definition of the object guarantees that the object is always initialized before use (the concern seems to be more oriented towards ensuring an object does not have an indeterminate value than that it have the correct value, which it might not be possible to assign at the point of definition).

One potential advantage, in providing an initializer, is that if the definition is moved from an inner to an outer scope (moving from an outer to an inner scope will cause a diagnostic to be issued for any references from outside the new scope), during program modification, its initialization is moved at the same time (if the initial value is assigned in a statement, then that statement also needs to be moved; forgetting to do this is a potential source of faults). The following lists several potential disadvantages to this usage:

- The values of the operands in the expression providing the initial value may not be correct, at the point in the source where the initializer occurs. It is also possible that the object represents some temporary value that depends on the value of other objects defined in the same scope.
- A strong case can be made for initializing objects close to their point of use, so the associated source forms a readily comprehensible grouping (this is also an argument for moving the definition closer to its point of use). This issue is discussed in more detail elsewhere.

statement 1707 syntax

- Use of an initializer has been found to sometimes give developers a false sense of security, causing the assignment of a value to be overlooked. For instance, when two very similar sequences of source code occur in a function the second is often created by modifying a copy of the first one. A commonly seen mistake, when initializers are used, is to forget to give some object a new initial value.

```
1  struct T;
2  extern struct T *p_list;
3
4  void f(void)
5  {
6  struct T *walk_p_list = p_list;
7
8  while (walk_p_list)
9     {
10 /* Walk p_list doing something. */
11    }
12
13 /*
14  * Copied above loop, but overlooked giving walk_p_list an initial value.
15  */
16 while (walk_p_list)
17    {
18 /* Walk p_list doing something similar (but loops cannot be merged). */
19    }
20 }
```

In C99 it is possible to intermix declarations and statements. This means that the point of declaration, of an object, could be moved closer to where it is first used. The issue of where to declare objects is discussed elsewhere.

identifier 1348 definition close to usage

There is no evidence to suggest that the benefits of unconditionally providing an initializer in the definition of objects are greater than the costs. For this reason no guideline recommendation is given.

---

unnamed members initialization of

Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. 1650

Commentary

This is explicitly stated otherwise in one other instance and is discussed there. The fact that a member is unnamed suggests the author did not intend any value it might happen to contain to be accessed. Providing a mechanism to specify their initial value has no obvious benefit.

1663 initializing including unnamed members

C90

The C90 Standard wording "All unnamed structure or union members are ignored during initialization." was modified by the response to DR #017q17.

C++

This requirement is not explicitly specified in the C++ Standard.

1651 Unnamed members of structure objects have indeterminate value even after initialization.

Commentary

Many developers are aware of the common implementation practice, if a sufficient number of members are initialized to zero, of zeroing all of the storage occupied by an object having a structure type. This C sentence points out that as far as the abstract machine is concerned unnamed members are not affected by the initializer in a definition.

184 abstract machine C

C90

This was behavior was not explicitly specified in the C90 Standard.

C++

This behavior is not explicitly specified in the C++ Standard.

Common Implementations

For objects having an aggregate type and where many members are initialized to zero, many implementations generate machine code to loop over the entire object, setting all bytes to zero; the nonzero values are then individually assigned.

1652 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

object value indeterminate

Commentary

This is a conceptual indeterminate value, an implementation is not required to create and assign it (when the declaration of such an object is encountered during program execution).

75 indeterminate value

```
1   extern _Bool g(void);
2
3   void f (void)
4   {
5   loop: ;
6   int i = 42; /* Explicit initialization occurs every time declaration is encountered */
7   int j;      /* as does implicit initialization to indeterminate value. */
8   if (g())    /* j always has an indeterminate value here. */
9      return;
10  i = 69;     /* Change value of i.   */
11  j = 0;      /* Assign a value to j. */
12  goto loop;
13  }
```

The behavior resulting from reading the value of an object having an indeterminate value depends on the type of the object. Objects having type **`unsigned char`** are guaranteed to contain a representable value (or an object having type array of **`unsigned char`**). In this case a read access results in unspecified behavior. The indeterminate value of objects having other types may be a trap representation (accessing an object having such a value results in undefined behavior).

571 unsigned char pure binary

581 trap representation

Other Languages

Some languages (e.g., Perl) implicitly assign a value to all objects before they are explicitly assigned to (in the case of Perl this is the value `undef`, which evaluates to either 0 or the zero length string). Some languages (e.g., Ada and Extended Pascal) allow the definition of a type to include an initial value that is implicitly assigned to objects defined to have that type.

Common Implementations

The value of such objects is usually the bit pattern that happened to exist in the storage allocated at the start its lifetime. This bit pattern may have been created by assigning a value to an object whose lifetime has ended, or the storage may have been occupied by more than one object, or it may have been used to hold housekeeping information. A few implementations zero the storage area, reserved for defined objects, on function entry.

Example

```
1  { float f=1.234; }
2  { int i;  /* i will probably be allocated storage previous occupied by f */ }
```

---

static initialization default value

If an object that has static storage duration is not initialized explicitly, then: 1653

Commentary

In the past many implementations implicitly initialized the storage occupied by objects having static storage duration to all bits zero prior to program startup. This practice is codified in the C Standard here; where the intent of *all bits zero*, i.e., a value representation of zero, is specified. The issue of initializing objects with static storage duration is discussed elsewhere.

static storage duration 151
initialized before startup
static storage duration 456
when initialized

C++

The C++ Standard specifies a two-stage initialization model. The final result is the same as that specified for C.

8.5p6 *The memory occupied by any object of static storage duration shall be zero-initialized at program startup before any other initialization takes place. [Note: in some cases, additional initialization is done later.]*

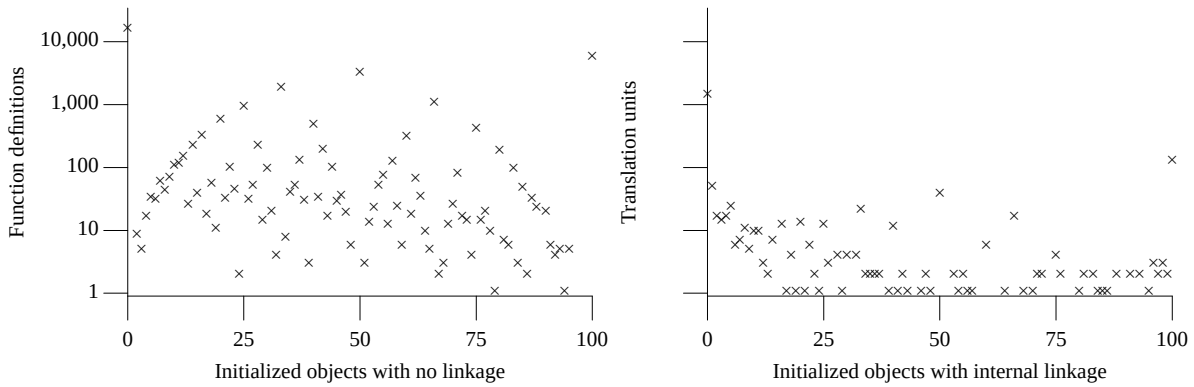


Figure 1652.1: Number of object declarations that include an initializer (as a percentage of all corresponding object declarations), either within function definitions (functions that did not contain any object definitions were not included), or within translation units and having internal linkage (while there are a number of ways of counting objects with external linkage, none seemed appropriate and no usage information is given here). Based on the translated form of this book's benchmark programs.

Other Languages

Most other languages do not treat one kind of uninitialized objects any differently than another kind.

Coding Guidelines

The behavior described in the following sentences is common developer knowledge. There is no obvious benefit in recommending against making use of it, on the basis that all behavior should be explicit.

1654 — if it has pointer type, it is initialized to a null pointer;

Commentary

The null pointer is also the only pointer value that is compatible with all pointer types.

749 null pointer
750 null pointer conversion yields null pointer

1655 — if it has arithmetic type, it is initialized to (positive or unsigned) zero;

Commentary

Zero is the constant most frequently assigned to an object and the most commonly occurring constant literal in source code. It is the initial value most likely to be chosen by a developer, if one had to be explicitly supplied.

825 integer constant usage

C90

The distinction between the signedness of zero is not mentioned in the C90 Standard.

Common Implementations

In most implementations this value is represented by all bits zero for all arithmetic types.

1656 — if it is an aggregate, every member is initialized (recursively) according to these rules;

member initialized recursively

Commentary

In the case of array types every element is assigned the same value.

Common Implementations

Most implementations treat an aggregate, that is being implicitly initialized, as a single entity. That is the most efficient way of assigning all bits zero is used.

1657 — if it is a union, the first named member is initialized (recursively) according to these rules.

Commentary

Having decided to support the initialization of objects having a union type, the specification either had to provide a mechanism for denoting a member to be initialized, or provide a rule to enable readers to deduce the member that will be initialized. The rule that the first named member is initialized fit in with the general English (and perhaps other cultures) convention of starting at the top and working down (rather than starting at the bottom and working up). It is also less likely to cause maintenance problems (since developers tend to add new members at the end of the list of current members).

If the first named member is an aggregate all the members of that aggregate are initialized.

C90

This case was not called out in the C90 Standard, but was added by the response in DR #016.

C++

The C++ Standard, 8.5p5, specifies the first data member, not the first named data member.

1658 The initializer for a scalar shall be a single expression, optionally enclosed in braces.

initializer scalar

Commentary

Allowing initializers for scalars to be enclosed in braces can simplify the automatic generation of C source (the generator does not need any knowledge of the type being initialized, it can always output braces). Braces are not optional when a value appears as the right operand of an assignment operator.

C++

8.5p13 *If T is a scalar type, then a declaration of the form*

```
T x = { a };
```

is equivalent to

```
T x = a;
```

This C++ specification is not the same the one in C, as can be seen in:

```
struct DR_155 {
                int i;
               } s = { { 1 } }; /* does not affect the conformance status of the program */
                                // ill-formed
```

8.5p14 *If the conversion cannot be done, the initialization is ill-formed.*

While a C++ translator is required to issue a diagnostic for a use of this ill-formed construct, such an occurrence causes undefined behavior in C (the behavior of many C translators is to issue a diagnostic).

Other Languages

Most languages that support initializers do not allow redundant braces to be used for scalars.

Coding Guidelines

Initializers for objects having scalar type are rarely enclosed in braces. For this reason they are not discussed further here.

The initial value of the object is that of the expression (after conversion); 1659

Commentary

The conversion is to the type of the object being initialized.

C90

The C90 wording did not include "(after conversion)". Although all known translators treated initializers just like assignment and performed the conversion.

---

initializer type constraints

the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type. 1660

Commentary

simple assignment constraints 1296

simple assignment 1303

The unqualified type needs to be specified because the constraints for simple assignment do not permit object having a const-qualified type to be assigned a value. A consequence of taking the unqualified type is that initialization does not have exactly the same semantics as simple assignment if the object has a volatile-qualified type.

C++

Initialization is not the same as simple assignment in C++ (5.17p5, 8.5p14). However, if only the constructs available in C are used the behavior is the same.

Coding Guidelines

simple assignment constraints 1296

The applicable guideline recommendation are those that apply to simple assignment.

1661 The rest of this subclause deals with initializers for objects that have aggregate or union type.

Commentary

The standard defines additional syntax and semantics for initializers of objects having aggregate or union type.

1662 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type.

Commentary

Just like simple assignment, it is possible to initialize a structure or union object with the value of another object having the same type. In the case of a union object the value assigned need not be that of the first named member.

C++

The C++ Standard permits an arbitrary expression to be used in all contexts (8.5p2).

This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation. C++ translators that have an *operate in C mode* switch do not always diagnose initializers that would not be acceptable to all conforming C translators.

1663 In the latter case, the initial value of the object, including unnamed members, is that of the expression.

initializing including unnamed members

Commentary

The former case is the subject of discussion of most of the rest of the C subclause. There is one case where unnamed members participate in initialization.

1650 unnamed members initialization of

```
1  #include <stdio.h>
2
3  struct T_1 {
4              int mem_1;
5              unsigned : 4;
6              double mem_2;
7             };
8  struct T_2 {
9              int mem_1;
10             unsigned int mem_name: 4;
11             double mem_2;
12            };
13 union T_3 {
14            struct T_1 su;
15            struct T_2 sn; /* Both structure types have a common initial sequence. */
16           } gu;
17 struct T_1 s;
18
19 int main(void)
20 {
21 union T_3 lu_1 = {{0, 0.0}};
22 /*
23  * The value of lu_1.sn.mem_name is unspecified here.
24  */
25 gu.sn.mem_name = 1;
26 union T_3 lu_2 = gu;
27
28 if (lu_2.sn.mem_name != 1)
29    print("This is not a conforming implementation\n");
30 }
```

C++

The C++ Standard does not contain this specification.

---

initialize array of char

An array of character type may be initialized by a character string literal, optionally enclosed in braces. 1664

Commentary

A string literal is represented in storage as a contiguous sequence of characters, an array is a contiguous sequence of members. This specification recognizes parallels between them. The rationale for permitting optional braces is the same as that for scalars.

initializer scalar 1658

Coding Guidelines

What are the cost/benefit issues of expressing the initialization value using a string literal compared to expressing it as a comma separated list of values? The string literal form is likely to have a more worthwhile cost/benefit when:

- the value is likely to be familiar to readers as a character sequence (for instance, it is a word or sentence). There is a benefit in making use of this existing reader knowledge, or
- the majority of the individual characters in the string literal are represented in the visible source using characters, rather than escape sequences, the visually more compact form may require less effort, from a reader, to process.

One situation where use of a string literal may not be cost effective is when the individual character values are used independently of each other. For instance, they represent specific data values or properties. In this case it is possible that macro names have been defined to represent these values. Referencing these macro names in the initializer eliminates the possibility that a change to their value will not be reflected in the initializer.

---

initialize uses successive characters from string literal

Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array. 1665

Commentary

The declaration:

```
unsigned char uc[] = "\xFF";
```

is equivalent to:

```
unsigned char uc[2] = { (unsigned char)(char) 0xFF, 0 };
```

C++

The C++ Standard does not specify that the terminating null character is optional, as is shown by an explicit example (8.5.2p2).

An object initialized with a string literal whose terminating null character is not included in the value used to initialize the object, will cause a diagnostic to be issued by a C++ translator.

```
char hello[5] = "world"; /* strictly conforming */
                         // ill-formed
```

Common Implementations

Some implementations store the string literal as part of the program image and copy it during initialization. Others generate machine code to store constants (the individual character values, often concatenated to form larger constants, reducing the number of store instructions) into the object.

Coding Guidelines

Developers sometimes overlook a string literal's terminating null character in their calculation of the number of array elements it will occupy (either leaving insufficient space to hold the null character, when an array size is specified, or forgetting that storage will be allocated for one, in the case of an incomplete array type). There is no obvious guideline recommendation that might reduce the probability of a developer making these kind of mistakes.

Example

```
char a_1[3] = "abc", /* storage holds | a | b | c |         */
     a_2[4] = "abc", /* storage holds | a | b | c | 0 |     */
     a_3[5] = "abc", /* storage holds | a | b | c | 0 | 0 | */
     a_4[]  = "abc"; /* storage holds | a | b | c | 0 |     */
```

In (assuming any undefined behavior does not terminate execution of the program):

```
unsigned char s[] = "\x80\xff";
```

the first element of `s` is assigned the value `(unsigned char)(char)128` and the second element the value `(unsigned char)(char)255`.

1666 An array with element type compatible with **`wchar_t`** may be initialized by a wide string literal, optionally enclosed in braces.

initialize array of wchar_t

Commentary

The applicable issues are the same as for an array of character type.

1664 initialize array of char

Coding Guidelines

The cost/benefit of using a wide string literal in the visible source, rather than a comma separated list, will be affected by the probability that the wide characters will appear, to a reader, as a glyph rather than some multibyte sequence (or other form of encoding).

1664 initialize array of char

1667 Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

initialize uses successive wchar_t from string literal

Commentary

The applicable issues are the same as for an array of character type

1665 initialize uses successive characters from string literal

---

1668 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.

Commentary

The syntax requires a list of initializers to be brace enclosed (the braces serve to disambiguate what would otherwise look like a declarator list). While the braces are not strictly necessary for union types (there is only one initializer), their visual appearance is consistent with braces being used in structure and union type definitions.

Other Languages

Many languages support this form of initializer (although the delimiters are not always braces).

1669 Each brace-enclosed initializer list has an associated *current object*.

current object brace enclosed initializer

Commentary

This introduces the term *current object*. This terminology is new in C99 and is not commonly used by developers. Current objects are associated only with brace-enclosed initializer lists.

footnote 1685 127

A lot of background knowledge of how "things are supposed to work" is needed to understand out how the *current object* maps to the object being initialized. Some of this knowledge is encoded in the extensive examples provided in the Standard.

C90

The concept of *current object* is new in C99.

C++

The concept of *current object* is new in C99 and is not specified in the C++ Standard. It is not needed because the ordering of the initializer is specified (and the complications of designation initializers don't exist, because they are not supported):

8.5.1p2 *. . . , written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate.*

Other Languages

Fortran does not restrict the initializer list to providing initial values for one object. Its **DATA** statement may contain a list of separate, unrelated, objects followed by a list of values (used to initialize the separate objects at program startup).

---

initialization no designator member ordering

When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.[127)] 1670

Commentary

This selection of mapping between members and initializers is the one that is most consistent with what people are likely to expect to occur.

C90

Otherwise, the initializer for an object that has aggregate type shall be a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order; and the initializer for an object that has union type shall be a brace-enclosed initializer for the first member of the union.

The wording specifying named member was added by the response to DR #016.

Other Languages

This convention is common to most languages that support some form of initializer.

Coding Guidelines

An object declaration, whose derived type includes a structure type, that includes an initializer where no designations are present contains an ordering dependency between the structure members and the values given in the initializer. For instance, the following definition contains a member ordering assumption that is not guaranteed by the C Standard:

EXAMPLE 1702 div_t

```
1  #include <stdlib.h>
2
3  div_t val = {3, 4};
```

initialization using a designator

Is there worthwhile cost/benefit in a guideline recommendation specifying that initialization values corresponding to structure members always include a designator? Designators are not supported by C++ or C90, so such a recommendation would be C99 specific. The following are some of the potential benefits of using designators in the initialization lists of objects having a structure type:

- Removing the dependency between members and their corresponding initialization value. This benefit is only realized if the ordering of existing members changes (adding new members after the existing members does not change the existing dependencies). The possibility of developers writing code that contains dependencies on member ordering is appreciated by developers (and third-party library vendors) and the cost of changes to existing code (customer complaints) are factored into any consideration of changing structure member order. Experience shows that new members are usually added to the end of existing structure types.
- As a memory aid for the reader of the initializer. While comprehending an initializer readers have to process its components, while remembering information about their position within the being initialized. Having a designator visible provides a mechanism for readers to for holding and refreshing information in the visuo-spatial sketch pad, freeing up other working memory resources for other tasks.

0 visuo-spatial memory
0 memory developer

Given the existing practice of adding new members to the end of an existing structure type (the C++/C90 compatibility issue could be solved via the use of a macro) there does not appear to be sufficient benefit for a guideline recommending that designators always be used.

Initialization of objects having an array type involves a (potentially) large number of values of the same type. Experience shows that the value of many elements is often zero. Designators provide a method of reducing the number of zeroes that appear in the source. However, specifying the conditions under which there is a worthwhile cost/benefit for their usage is likely to involve a complex calculation. At the time of this writing there is insufficient experience with the use of this construct to know whether simple rules can be specified. For this reason no guideline recommendation is given.

1671 In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator.

initialization using a declarator

Commentary

The designator explicitly specifies which member (of the current object) the initialization value refers to.

```
1   int a[5] = { [2] = 3 };
```

C90

Support for designations is new in C99.

C++

Support for designations is new in C99, and is not available in C++

Other Languages

Some languages (e.g., Ada) support an equivalent construct.

Coding Guidelines

Some of the costs and benefits of using designators are discussed elsewhere.

1670 initialization no designator member ordering

1672 Initialization then continues forward in order, beginning with the next subobject after that described by the designator.[128)]

Commentary

The designation can be thought of as a "goto" within the current object. Once a designator has specified a member at which initialization is to occur, subsequent initializations continue from that point until another designator or the end of the type of the current object is reached.

Other Languages

Ada requires that initializers either be written without using any designators or be written only using designators (i.e., no mixing).

Coding Guidelines

initialization 1670
no designator
member ordering

Is the use of designators an all or nothing choice? This question is a special case of the member ordering dependency issue discussed elsewhere. At the time of this writing there is not sufficient experience available with the use of this construct to be able to reliably make any recommendations.

In the two definitions:

```
int a1[] = {
             [10] = 5, 9, 3,
             [23] = 1, 0, 16,
            };
int a2[] = {
             [10] = 5, [11] = 9, [12] = 3,
             [23] = 1, [24] = 0, [25] =16,
            };
```

it is possible to imagine the following two scenarios— where modifications to a program:

- require the three initial values starting at `[10]` need to be offset by four elements, to `[14]`. In this case there is greater potential for mistakes being introduced by incorrectly modifying (or not modifying) the two following designators, in the initializer for `a2`, or
- require that the initial value at `[10]` be moved by four elements to `[14]`. In this case there is greater potential for mistakes being introduced by failing to create a designator for `[11]` (or incorrectly specifying it), in the initializer for `a1`.

---

designator list
current object

Each designator list begins its description with the current object associated with the closest surrounding brace pair. 1673

Commentary

Here usage of the term *designator list* refers to the syntactic terminal of that name. The member specified by a designator is context dependent. While it is not necessary to specify the path from the outermost object to the current object, it is necessary to specify a path to any nested subobject being initialized (or use nested braces). An initializer enclosed in a brace pair is used to initialize the members of a contained subaggregate or union.

initializer 1679
brace pair

C90

Support for designators is new in C99.

C++

Support for designators is new in C99, and is not available in C++

Coding Guidelines

It is possible to write initializers without using any nested brace pair. However, the additional cost of inserting pairs of braces into an initializer is minimal and the potential benefit is large. These benefits include the following:

- Providing a visual aid that can be used, in conjunction with white space and new line characters, to highlight individual sequences of initializers. In particular the appearance of an opening brace helps disambiguate whether initializers at the start of a line are a continuation of the initializers from the previous line, or denote the start of the initializers for a different object. A closing brace provides explicit visual information that any following initializers belong to a different subobject.

- An increased likelihood of developer mistakes (e.g., a missing initializer value or unintended additional initializer value) resulting in diagnostics being generated by translators. It is possible to specify fewer initializers than there are objects to initialize, and many initializers and objects have an integer type (which reduces the probability of a mismatched initializer value and object having an incompatible type, i.e., no diagnostic will be generated). An opening brace may only appear when the current object has an aggregate or union type, and translators are required to perform this check. There is a limit to the number of initializers that can occur between matching braces (the number of subobjects being initialized) and translators are required to perform this check.

Enclosing initializers in matching braces isolates them from other initializers for other members. Such usage also enables translators to perform a finer grained level of checking on the expected number of initializers for each member.

Cg 1673.1 Any optional braces shall not be omitted in an initializer for an aggregate or union type.

Example

In the following, the initializers all assign the same value to the same members:

```
struct S {
           char m_1_1;
           struct {
                   char m_2_1;
                   struct {
                           char m_3_1;
                           char m_3_2;
                          } m_2_2;
                   char m_2_3;
                  } m_1_2;
           char m_1_3;
          };

struct S x_1 = {'a',           'b', 0,             'c', 0, 'd'};
struct S x_2 = {'a', .m_1_2 = {'b',      {.m_3_2 = 'c'} }, 'd'};
struct S x_3 = {'a',          {'b',            {0, 'c'} }, 'd'};
struct S x_4 = {'a', {.m_2_1 = 'b', .m_2_2.m_3_2 = 'c'},   'd'};
```

1674 Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[129)]

Commentary

Each designator list is independent of any other designator list and any change of current object only applies during the evaluation of a given designator list.

1689 designator list independent

Coding Guidelines

The cost/benefit issues associated with using designators are discussed elsewhere. This coding guideline subsection discusses whether any guideline recommendations on the forms of usage, if it was decided to use them in initializers, might be cost effective.

1670 initialization no designator member ordering
1670 initialization using a designator

There may be a reason for automatically generated source to initialize members in what appears to be a random order. Having a designator list specify members in the same order as they occur in the declaration of the aggregate type is a consist mapping in many cultures and is also consistent with the order used when no designators are present. However, one of the benefits of using designators is that the relative order of values within an initializer is independent of the order used in the declaration of an aggregate.

1670 initialization no designator member ordering

There is no evidence to suggest that (designators are a new construct and at the time of this writing little experience has been gained in how they are used), in practice, developers will commonly order designators in any way other than an order that matches the order of the member in the declaration of the aggregate type (or at least the one visible at the time the code was first written, which may subsequently change). Until more experience has been gained in how developers use this construct there is no point in idle speculation over what guideline recommendations might be appropriate.

Example

In the following the initializers all assign the same values to the same members:

```
1  struct S {
2            char m_1_1;
3            struct {
4                   char m_2_1;
5                   struct {
6                             char m_3_1;
7                             char m_3_2;
8                            } m_2_2;
9                   char m_2_3;
10                  } m_1_2;
11           char m_1_3;
12           };
13
14 struct S x_1 = {'a',                'b', 0,             'c'      , 0, 'd'};
15 struct S x_2 = {'a',      {.m_2_1 = 'b', .m_2_2.m_3_2 = 'c'},         'd'};
16 struct S x_3 = {'a', .m_1_2.m_2_1 = 'b', .m_1_2.m_2_2.m_3_2 = 'c', 0, 'd'};
```

The current object that results at the end of the designator list is the subobject to be initialized by the following initializer. 1675

Commentary

The following initializer can be a brace enclosed initializer.

```
1  struct S {
2            char m_1_1;
3            struct {
4                   struct {
5                          char m_3_1;
6                          char m_3_2;
7                          } m_2_1;
8                   char m_2_2;
9                   } m_1_2;
10           };
11 struct S x_1 = {.m_1_2 =       {1, 2,  0}};
12 struct S x_2 = {.m_1_2 =     { {1, 2}, 0}};
13 struct S x_3 = {.m_1_2.m_2_1 = {1, 2}};
```

---

initialization in list order

The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[130)] 1676

Commentary

Objects with automatic storage duration are not required to have initializers that are constant expressions. Expressions can cause side effects. While this requirement specifies the order in which initialization occurs it does not specify the order in which the initializers are evaluated. The following declaration:

footnote 130 1690

```
union { char x[5]; } u = { .x[1] = 7 };
```

is a shorthand for:

```
union { char x[5]; } u = { .x = { [1] = 7 } };
```

Expanding out shorthand forms can help make it easier to deduce the initial values of some members. For instance, in:

```
union {
       char a[5];
       char b[5];
      } u = {
              .a[1] = 7,
              .b[2] = 8
             };
```

it might be thought that the value of `u.b[1]` is 7. However, writing it out in non-shorthand form we get:

```
union {
       char a[5];
       char b[5];
      } u = {
             .a = { [1] = 7 },
             .b = { [2] = 8 }
             };
```

where it is easily seen that the value of `u.b[1]` is 0.

C++

The C++ Standard does not support designators and so it is not possible to specify more than one initializer for an object.

Other Languages

Ada does not permit more than one initializer to be specified for the same subobject.

Common Implementations

It is too early to know whether it is worthwhile for optimizers to invest in looking for and removing overridden initializers. Initializers that are overridden and do not generate side effects or are depended on by other initializers may be removed general dead code elimination optimizations.

Coding Guidelines

The guideline recommendation dealing with the evaluation ordering between sequence points is applicable here. Providing an initializer for an object that is subsequently overridden might be treated as suspicious for several reasons, including the following:

187.1 sequence points all orderings give same value
1704 EXAMPLE overriding values

- A reader may only see the first initializer value and not any later ones, leading to an incorrect interpretation of program behavior.
- Evaluation of the initializer may be expected to cause side effects, or be used as an operand in the initializer for another subobject. Not evaluating an initializer, that is overridden, may have unexpected consequences.

```
extern int glob;
struct S {
          int m_1;
          int m_2;
          int m_3;
          };
```

```
7
8   void f(void)
9   {
10  struct S loc = {
11                  .m_1 = glob++,
12                  .m_2 = loc.m2+1,
13                  .m_1 = 1
14                  };
15  }
```

dead code 190

- The overridden initializer is essentially dead code. This issue is discussed elsewhere.
- The usage might be considered suspicious, especially if another member had not been explicitly initialized.

Designations are new in C99 and it is too early to know whether a specific guideline recommending against their use is worthwhile (there are many constructs guidelines could recommend against, but don't because they rarely appear in source code). The conclusions of the discussion on dead code may be sufficient.

dead code 190

---

object initialized but not explicitly

all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration. 1677

Commentary

The presence of an initializer in an object definition causes different behavior than the absence of an initializer. If the definition includes an initializer, even if it only initializes a single member, all members are given a known value. However, all members of an object defined without an initializer (and not having static storage duration) have an indeterminate value, at the point of definition.

object 463 indeterminate each time declaration reached

The implicit value assigned to objects that have static storage duration is zero (or null).

static initialization 1653 default value

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If constructs that are only available in C are used the behavior is the same as that specified in the C Standard.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require that initializers be specified for all subobjects.

Common Implementations

Two implementation strategies are to zero all members of the initializer (using a loop) and then assigning specific values to specific members, or to assign specific values (which may include the implicit values) to all members. The better optimizers perform a cost/benefit analysis to select the strategy to use (which for large objects may involve applying different ones for different subobjects).

Coding Guidelines

coding guidelines 0 other documents

A general principle promoted in many coding guideline documents is that all operations should be explicit. However, explicitly specifying zero values for members of an aggregate object has a number of costs that the implicit usage does not, including the following:

- A large number of zeros in the visible source increase the effort needed, by readers, to locate the nonzero values. The zeros have become visual noise, that do not provide information to readers.
- Increased maintenance costs. Having to update the initializer list every time the aggregate type changes (either because of a change in the number of elements, or the number of members). If designators are used member names may also need to be updated.

Given these costs and the fact that developers are generally aware of the default behavior, there does not appear to be a worthwhile benefit in a guideline recommending that the behavior be made explicit.

1678 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions.

Commentary

There are no special cases that apply to nested aggregates or unions.

1679 If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union.

initializer brace pair

Commentary

The brace pair provide a mechanism for explicitly specifying which initializers belong to a subaggregate or contained union. Only the initializers enclosed by the braces are used to initialize its corresponding subaggregate or union. If there are fewer initializers than members the remaining members are implicitly initialized. Each brace-enclosed initializer list also has an associated current object.

1669 current object brace enclosed initializer

Other Languages

Fortran uses parenthesis and forward slash (e.g., `(/.../)` and `/.../`), Ada uses parenthesis (e.g., `(...)`) and Extended Pascal uses square brackets (e.g., `[...]`).

Coding Guidelines

The guideline recommendation that braces always be used is discussed elsewhere.

1673.1 initializer use braces

1680 Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union;

Commentary

If the initializer values are not enclosed by a matching pair of braces, values from the current list of initializers are used.

Other Languages

The Fortran `DATA` statement also specifies this behavior. However, most languages that support some form of initialization, associated with an objects declaration, require stricter correspondence between initializer value and subobject being initialized.

Coding Guidelines

The coding guideline issues associated with this form of initializer are discussed elsewhere.

1673.1 initializer use braces

1681 any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

Commentary

That is, any remaining initializers up until the matching right brace.

C90

This behavior was not pointed out in the C90 Standard.

1682 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

initializer fewer in list than members

Commentary

This wording spells out particular cases, removing the possibility of other interpretations being applied to the general wording on implicit initialization given earlier.

1677 object initialized but not explicitly

C90

The string literal case was not explicitly specified in the C90 Standard, but was added by the response to DR #060.

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If only constructs available in C are used the behavior is the same as that specified in the C Standard.

Other Languages

Many other languages (e.g., Ada and Extended Pascal) do not explicitly specify a behavior for this case.

Coding Guidelines

object initialized but not explicitly 1677

The coding guideline discussion given elsewhere is applicable here.

---

array of unknown size initialized

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. 1683

Commentary

An initializer provides another method of specifying the size (number of elements) of an array type.

C90

If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements.

Support for designators is new in C99.

Other Languages

Ada specifies a similar rule for deducing both the lower and upper bounds of an array.

Coding Guidelines

Specifying the size of the array via the contents of an initializer reduces the amount of effort needed to write the array definition. In this case the number of elements in the array has to be obtained using a constant expression of the form `(sizeof(arr) / sizeof(arr[0]))`. However, in some cases developers do not specify a size between the **`[]`** tokens, even although the value is readily available to them. Such usage becomes a potential cause of maintenance problems if two unrelated mechanisms are used to denote the number of array elements (e.g., some numeric literal, and an expression using **`sizeof`**). Using the known size value in the declaration of the array provides some degree of checking (i.e., translators are required to issue a diagnostic if too many initializers are specified, but not if too few are specified).

Example

```
#define A_SIZE 9

int a_1[]       = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int a_2[A_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; /* Diagnostic issued. */
int a_3[A_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7      }; /* Diagnostic not issued. */
```

---

initializer completes incomplete array type

At the end of its initializer list, the array no longer has incomplete type. 1684

Commentary

The initializer provides all of the information needed to complete the type. However, the single pass translation nature of C means that the completion of the type does not affect any constraint requirements that apply to constructs appearing earlier in the processing of the declaration. For instance, in: 10 implementation single pass

```
int a[][] = {{1}, {2}, {3}};    /* Only one array size can be deduced from initializer. */
int b[sizeof(b)*2] = {1, 2, 3}; /* Operand of sizeof has incomplete type when it is processed. */
```

C++

The C++ Standard does not specify how an incomplete array type can be completed. But the example in 8.5.1p4 suggests that with an object definition, using an incomplete array type, the initializer creates a new array type. The C++ Standard seems to create a new type, rather than completing the existing incomplete one (which is defined, 8.3.4p1, as being a different type).

1685 127) If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists. footnote 127

Commentary

```
struct S {
          int m_1_1;
          struct {
                 int m_2_1;
                 int m_2_2;
                 } m_1_2;
          int m_1_3;
          };

struct S x_1 = {1,
                2, /* Current object does not become m_1_2 */
        .m_2_2 = 3, /* Constraint violation, designator should be .m_1_2.m_2_2 */
                4
               };
```

C90

The concept of current object is new in C99.

C++

The concept of current object is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

Developer misunderstandings, in this case, about what constitutes the current object could be harmless in that incorrect use of designators causes a diagnostic to be issued. However, the same member name appearing on its own (e.g., `.next`) in more than one designator is a possible source of reader visual confusion. One way of reducing the possibility of visual confusion is to use additional members in designator (e.g., `.left.next` and `.right.next`). But this usage is dependent on which subaggregate the current member denotes, which will be affected by the brace nesting. Until more experience has been gained in the use of designators it is not possible to estimate the cost/benefit trade-offs involved in using braces or designators containing more member selections.

1686 128) After a union member is initialized, the next object is not the next member of the union; footnote 128

Commentary

Only one member of a union is required to hold a value at any time. 563 footnote 37

C90

The C90 Standard explicitly specifies, in all the relevant places in the text, that only the first member of a union is initialized.

---

instead, it is the next subobject of an object containing the union. 1687

Commentary

member initialized recursively 1656

Or, if that is a union or if the last member of an aggregate has just been processed, the next subobject of the containing type (and so on returning through any recursion) .

---

footnote 129

129) Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. 1688

Commentary

A designator cannot, for instance, denote a member of an object contained in the type of a member outside of the surrounding brace.

```
1  struct S {
2            int m_1_1;
3            struct {
4                    int m_2_1;
5                    int m_2_2;
6                   } m_1_2;
7            int m_1_3;
8           };
9
10 struct S x_1 = {1,
11                   {2,
12                    .m_1_3 = 3 /* Constraint violation. */
13                   }
14                 };
```

Coding Guidelines

Any attempt to specify, in a designator, a member that is not in the strict subobject will cause a diagnostic to be generated.

---

designator list independent

Note, too, that each separate designator list is independent. 1689

Commentary

current object 1669 brace enclosed initializer

Each separate designator list is independent in the sense that a designator list does not change the current object in the way that a brace-enclosed initializer does.

```
1  struct S {
2            int m_1_1;
3            struct {
4                    int m_2_1;
5                    int m_2_2;
6                   } m_1_2;
7           };
8  struct S x_1 = {
9                   .m_1_2.m_2_1 = 1, /* Does not change the current object to be m_1_2. */
10                        .m_2_2 = 3  /* Constraint violation. */
11                 };
12 struct S x_2 = {
13                  .m_1_2 = {.m_2_1 = 1,  /* { Changes the current object to be m_1_2. */
14                            .m_2_2 = 3}  /* OK. */
15                 };
```

footnote 130

1690 130)Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

Commentary

This sentence was added by the response to DR #208.

DR #208

Committee Response

The question asks about the expression

```
int a [2] = { f (0), f (1), [0] = f (2) };
```

and the meaning of the wording

each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;

It was the intention of WG14 that the call `f(0)` might, but need not, be made when a is initialized. If the call is made, the order in which `f(0)` and `f(2)` occur is unspecified (as is the order in which `f(1)` occurs relative to both of these). Whether or not the call is made, the result of `f(2)` is used to initialize `a[0]`.

The wording of paragraph 23:

The order in which any side effects occur among the initialization list expressions is unspecified.

should be taken to only apply to those side effects which actually occur.

1691 The order in which any side effects occur among the initialization list expressions is unspecified.[131)]

Commentary

1714 full expression initializer

An initializer was defined to be a full expression in C90. However, support for nonconstant initial values is new in C99 and this sentence points out the consequences. While operators (the comma in an initialization list is a punctuator, not an operator) that have sequence points may occur in the initialization list, just like in an expression, there is no requirement that these sequence points occur in a particular order.

C90

The C90 Standard requires that the expressions used to initialize an aggregate or union be constant expressions. Whatever the order of evaluation used the external behavior is likely to be the same (it is possible that one or more members of a structure type are volatile-qualified).

C++

The C++ Standard does not explicitly specify any behavior for the order of side effects among the initialization list expressions (which implies unspecified behavior in this case).

Other Languages

Ada explicitly states that the order of evaluation is unspecified.

Coding Guidelines

187.1 sequence points all orderings give same value

The guideline recommendation dealing with the evaluation order between sequence points is applicable here. The extent to which developers will incorrectly believe the comma punctuators between designators is a sequence point is not known. Correcting any such a belief is an educational issue and is outside the scope of these coding guidelines. Because of the declaration nature of an initializer it is more difficult to break it up into smaller, independent, components (as is possible for expressions in a statement context). The extent to which this will lead to complicated and difficult to comprehend initializers is not known.

Example

```
#include <stdio.h>

void f(void)
{
int loc_1 [] = {printf("Hello"), printf("world")};
}
```

there are two possible character sequences that may be output, either **Helloworld**, or **worldHello**.

EXAMPLE 1 Provided that **<complex.h>** has been **#included**, the declarations 1692

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize **i** with the value 3 and **c** with the value *5.0 + i3.0*.

Commentary

In this example the macro I expands to a constant expression of type **const float _Complex**. The initialization value is actually equivalent to `5.0f+i3.0f`.

The wording was changed by the response to DR #293.

C90

Support for complex types is new in C99.

C++

The C++ Standard does not define an identifier named **I** in **<complex>**.

Coding Guidelines

integer 688 conversion to floating

Some of the issues applicable to this example are discussed elsewhere.

EXAMPLE 2 The declaration 1693

```
int x[] = { 1, 3, 5 };
```

defines and initializes **x** as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

Commentary

array element 1567 not incomplete type

An arrays element type is required to be an object type, not an incomplete type. This means it is not possible to use an initializer to specify the size of more than one array dimension.

```
int y[][] = { {1, 2}, {3, 4}, {5, 6} }; /* Constraint violation. */
```

EXAMPLE 3 The declaration 1694

```
int y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of **y** (the array object **y[0]**), namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise the next two lines initialize **y[1]** and **y[2]**. The initializer ends early, so **y[3]** is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y[0]** does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require the initializer to have a form similar to the first declaration and do not support a form equivalent to the second.

Coding Guidelines

The guideline recommendation that might be applicable to this example is the use braces to delimit initializers for members having an aggregate type.

1673.1 initializer use braces

1695 EXAMPLE 4 The declaration

```
int z[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **z** as specified and initializes the rest with zeros.

Other Languages

Some languages (e.g., Ada and Extended Pascal) require that initializers be specified for all subobjects.

1696 EXAMPLE 5 The declaration

EXAMPLE inconsistently bracketed initialization

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: **w[0].a[0]** is 1 and **w[1].a[0]** is 2; all the other elements are zero.

Commentary

An additional pair of braces is needed for the declaration:

```
1  struct { int a[3], b; } w[] = { { { 1 }, 2 } };
```

to define an array having a single element (i.e., b is initialized to 2). An example of an alternative form of initializer is given elsewhere.

1703 EXAMPLE designators with inconsistently brackets

1697 131) In particular, the evaluation order need not be the same as the order of subobject initialization.

footnote 131

Commentary

In the following:

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  int loc_1 [] = {2, loc_1[0]+1};
6  int loc_2 [] = {2, loc_2[0]++};
7  }
```

the element `loc_1[1]` is not guaranteed to be initialized with a value that is one greater than `loc_1[0]`. The initializer for `loc_2` exhibits undefined behavior because the same object is modified more than once between sequence points.

1714 full expression initializer

C++

The C++ Standard does explicitly specify the ordering of side effects among the expressions contained in an initialization list.

Common Implementations

While some implementations might evaluate each designator in an initializer in a first to last order, others might treats it as a sequence of independent simple assignments (relying on existing optimization routines within the translator to generate the best quality machine code).

sequence points 187.1
all orderings give same value

Coding Guidelines

The guideline recommendation dealing with expression evaluation is applicable here.

EXAMPLE 6 The declaration 1698

```
short q[4][3][2] = {
        { 1 },
        { 2, 3 },
        { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0]` does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for `q[1][0]` and `q[2][0]` do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
        1, 0, 0, 0, 0, 0,
        2, 3, 0, 0, 0, 0,
        4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
        {
                { 1 },
        },
        {
                { 2, 3 },
        },
        {
                { 4, 5 },
                { 6 },
        }
};
```

in a fully bracketed form.

Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

Coding Guidelines

designator list 1673
current object

The use of braces to delimit designator lists reduces the probability that changes to the size of any array will cause a fault to be generated (because the change is not reflected in the number of initializer values needed).

Developers are familiar with non-initialized elements implicitly being assigned a value of zero. Explicitly specifying trailing zeros in a designator list requires effort from readers to visually process them.

1699 EXAMPLE 7

One form of initialization that completes array types involves typedef names. Given the declaration

```
typedef int A[];        // OK - declared with block scope
```

the declaration

```
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

Commentary

An example similar to this was submitted as DR #010 against the C90 Standard.

C++

The C++ Standard does not explicitly specify this behavior.

1700 EXAMPLE 8 The declaration

EXAMPLE
array initialization

```
char s[] = "abc", t[3] = "abc";
```

defines "plain" **char** array objects **s** and **t** whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
     t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines **p** with type "pointer to **char**" and initializes it to point to an object with type "array of **char**" with length 4 whose elements are initialized with a character string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

Commentary

Initialization is a context where the implicit conversion of an array type to a pointer to its first element is dependent on the type of the object it initializes.

729 array converted to pointer

C++

The initializer used in the declaration of `t` would cause a C++ translator to issue a diagnostic. It is not equivalent to the alternative, C, form given below it.

Other Languages

A few languages (e.g., Pascal, provided the, stronger, type compatibility rules are met) allow strings to be assigned to an object having an array of character type.

Coding Guidelines

The issues associated with using a string literal to represent a sequence of character constants are discussed elsewhere.

1664 initialize array of char

1701 EXAMPLE 9 Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
        [member_two] = "member two",
        [member_one] = "member one",
};
```

Commentary

The only type associations created by this usage exist in the readers head.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Other Languages

Languages in the Pascal family allow the type of the indexing expression to be specified in the array type declaration. For instance:

```
1  TYPE
2        member_enum = (member_one, member_two);
3  VAR
4        nm[] : Array[member_enum] of char;
```

Coding Guidelines

initialization 1670 no designator member ordering

The issue of the order of designators used in member initialization is discussed elsewhere.

EXAMPLE div_t

EXAMPLE 10 Structure members can be initialized to nonzero values without depending on their order: 1702

```
div_t answer = { .quot = 2, .rem = -1 };
```

Commentary

Use of designators in this way removes the dependency between the order of values in an initializer and the order of members in a structure (which in the case of `div_t` is not specified).

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Coding Guidelines

initialization 1670 no designator member ordering

The issue of member initialization order is discussed elsewhere.

EXAMPLE designators with inconsistently brackets

EXAMPLE 11 Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood: 1703

```
struct { int a[3], b; } w[] =
       { [0].a = {1}, [1].a[0] = 2 };
```

Commentary

EXAMPLE 1696 inconsistently bracketed initialization

Initializers for an object having this type are discussed elsewhere.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

EXAMPLE overriding values

EXAMPLE 12 Space can be “allocated” from both ends of an array by using a single designator: 1704

```
int a[MAX] = {
       1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

In the above, if **`MAX`** is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

Commentary

A MAX value of less than five would be a constraint violation.

1642 initializer value not contained in object

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

Coding Guidelines

The discussion on initialization list order is applicable here.

1676 initialization in list order

1705 EXAMPLE 13 Any member of a union can be initialized:

```
union { /* ... */ } u = { .any_member = 42 };
```

Commentary

This form of initialization has the benefit that it is not dependent on the relative ordering of members within the union.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

1706 **Forward references:** common definitions **<stddef.h>** (7.17).

6.8 Statements and blocks

---

statement syntax

1707

statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Commentary

A statement is the basic unit of executable code. The term *statement header* is sometimes used to refer to the keyword (e.g., **if**, **for**) and the immediately following sequence of tokens between, and including, parentheses in a *selection-statement* or *iteration-statement*.

statement header

C++

In C++ local declarations are classified as statements (6p1). They are called *declaration statements* and their syntax nonterminal is *declaration-statement*.

Other Languages

Imperative languages use statements to specify a programs execution time control flow, while *functional* languages use expressions. Other language families use fundamentally different building blocks, for instance *Logic*, or *constraint-based* languages specify relationships and a program *execution* is an attempt to find values that meet these relationships.

Common Implementations

A number of implementations[42, 476, 515, 1314] have added extensions that provide support for either concurrent execution of groups of statements, or parallel operations on arrays of values. These extensions tend to be very dependent on particular hardware implementations (whose architecture is usually driven by the kinds of problems they have been designed to solve) and no single model of concurrency, or parallel execution, has achieved a significant market share (compared to the others).

Coding Guidelines

The discussion in the following two subsections is based on drawing a parallel between natural language sentences and source code statements (i.e., statements are the sentences of a programs source code; sequences of them are used to create a compound statement (paragraph) and sequences of these used to build a function definition, a subsection).[1707.1] The reason for drawing this parallel is the desire to make use of some of the findings from research on human sentence processing (there being no equivalent studies performed on source code statements). The following is a broad summary of these findings:

compound statement 1729 syntax
function definition 1821 syntax

- Over short time periods the syntax of what is read (i.e., words) is remembered, while over longer periods of time only the semantics (i.e., the meaning; which may differ between readers because of integration into their personal world model) is remembered.
- The measure of a persons working memory capacity that has the highest correlation with their performance in reading comprehension tasks is the reading span test.

reading span 1707

- The closer previously obtained information is (i.e., the more recently the sentence containing it was read) to the point it is referenced, the higher the probability that readers will correctly integrate it into the information extracted from what they are currently reading.

What is remembered

Having read a sentence, what does the reader remember about it? Studies[152] have found that over short time periods the syntax (i.e., words) is remembered, while over longer periods of time only the semantics (i.e., the meaning) is remembered. Readers might like to try the following test (based on Jenkins[666]). Part 1: A line at a time, (1) read the sentence on the left, (2) look away and count to five, (3) answer the question on the right, and (4) repeat process for the next line.

```
The girl broke the window on the porch.                    Broke what?
The hill was steep.                                        What as?
The cat, running from the barking dog, jumped on the table.  From what?
The tree was tall.                                         Was what?
The old car climbed the hill.                              What did?
The cat running from the dog jumped on the table.          Where?
The girl who lives next door broke the window on the porch.  Lives where?
The car pulled the trailer.                                Did what?
The scared cat was running from the barking dog.           What was?
The girl lives next door.                                  Who does?
The tree shaded the man who was smoking his pipe.          What did?
The scared cat jumped on the table.                        What did?
The girl who lives next door broke the large window.       Broke what?
The man was smoking his pipe.                              Who was?
The old car climbed the steep hill.                        The what?
The large window was on the porch.                         Where?
The tall tree was in the front yard.                       What was?
```

[1707.1] In practice a better parallel might be between statements and the *intonation units* commonly used in spontaneous spoken speech. Individual statements often contain a single piece of information and need to be considered in association with other statements, much like speech e.g., "hey . . . ya know that guy John . . . down the poolhall . . . he bought a Harley . . . if you can believe that.".

The car pulling the trailer climbed the steep hill.	Did what?
The cat jumped on the table.	Where?
The tall tree in the front yard shaded the man.	Did what?
The car pulling the trailer climbed the hill.	Which car?
The dog was barking.	Was what?
The window was large.	What was?

You have now completed part 1. Please do something else for a minute or so, before moving on to part 2 (which immediately follows).

Part 2: when performing this part, do not look at the sentences from part 1 above. Now, a line at a time, (1) read the sentence on the left, (2) if you think that sentence appeared as a sentence in part 1 write a number between one and five expressing your confidence (one expressing very little confidence and five expressive a lot of confidence in the decision) next to *old*, otherwise write a number representing your confidence level next to *new*, and (3) repeat process for the next line.

The car climbed the hill.	old___, new ___
The girl who lives next door broke the window.	old___, new ___
The old man who was smoking his pipe climbed the steep hill.	old___, new ___
The tree was in the front yard.	old___, new ___
The window was on the porch.	old___, new ___
The barking dog jumped on the old car in the front yard.	old___, new ___
The cat was running from the dog.	old___, new ___
The old car pulled the trailer.	old___, new ___
The tall tree in the front yard shaded the old car.	old___, new ___
The scared cat was running from the dog.	old___, new ___
The old car, pulling the trailer, climbed the hill.	old___, new ___
The girl who lives next door broke the large window on the porch.	old___, new ___
The tall tree shaded the man.	old___, new ___
The cat was running from the barking dog.	old___, new ___
The cat was old.	old___, new ___
The girl broke the large window.	old___, new ___
The car climbed the steep hill.	old___, new ___
The man who lives next door broke the window.	old___, new ___
The cat was scared.	old___, new ___

You have now completed part 2. Count the number of sentences you judged to be *old*.

The surprise is that all of the sentences are new.

While reading you abstracted and remembered the general ideas contained in the sentences (they are based on the four *idea sets*, (1) “The scared cat running from the barking dog jumped on the table.”, (2) “The old car pulling the trailer climbed the steep hill.”, (3) “The tall tree in the front yard shaded the man who was smoking his pipe.”, and (4) “The girl who lives next door broke the large window on the porch.”).

In a study by Bransford and Franks[152] each idea unit was broken down into sentences containing single ideas (e.g., “The cat was scared.”), two ideas (e.g., “The scared cat jumped on the table.”), three ideas (e.g., “The scared cat was running from the dog.”), and four ideas (e.g., “The scared cat running from the barking dog jumped on the table.”). The results (see Figure 1707.1) show that the greater the number of ideas units included in a sentence, the greater a subjects confidence that the sentence was previously seen (independently of whether it had been).

The issue of what people remember about what they have previously read is discussed in more detail elsewhere. 1821 function definition syntax

Processing single sentences

Individual sentence complexity[225] has a variety of effects on human performance. A study by Kintsch and Keenan[735] asked subjects to read single sentences, each containing the same number of words, but

propositional form

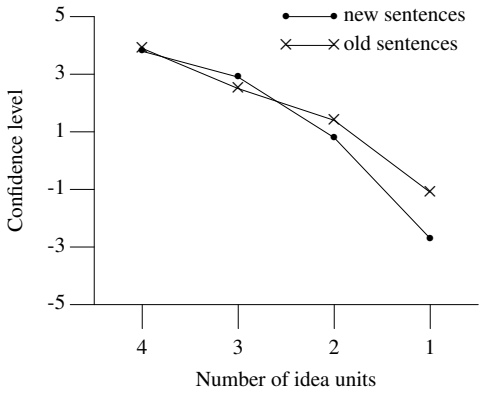
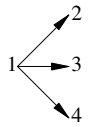


Figure 1707.1: Subject confidence level for having previously seen a sentence containing different numbers of idea units. Based on Bransford and Franks.[152]

Romulus, the legendary founder of Rome, took the women of the Sabine by force.

1 (took, Romulus, women, by force)
2 (found, Romulus, Rome)
3 (legendary, Romulus)
4 (Sabine, women)



Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.

1 (because, α, β)
2 $\alpha \rightarrow$ (fell down, Cleopatra)
3 $\beta \rightarrow$ (trust, Cleopatra, figures)
4 (foolish, trust)
5 (fickle, figures)
6 (political, figures)
7 (part of, figures, world)
8 (Roman, world)

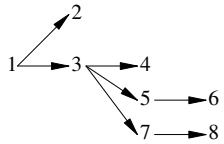


Figure 1707.2: Two sentences, one containing four and the other eight propositions, and their propositional analyses. Based on Kintsch and Keenan.[735]

varying in the number of propositions they contained (see Figure 1707.2). The time taken to read each sentence and recall it (immediately after reading it) was measured.

The results (see Figure 1707.3) show that reading rate decreases as the number of propositions in a sentence is increased (with the total number of words remaining the same). A later study[736] found that reader performance was also affected by the number of word concepts in a sentence and the grammatical form of the propositions (subordinate or superordinate).

Developers are often told to break up a complex statement or expression into several simpler ones. However, in many cases the quantity of interest is the comprehension cost of all of the code (it is possible that in some cases readers may only need to comprehend a subset of the simpler statements). It is therefore necessary to ask whether reader comprehension effort is minimized by having a single complex statement or having several simpler statements? While it is not yet possible to answer this question, some of the issues are known and the following subsection discusses the integration of information between related sentences.

statements integrating information between

Integrating information between sentences

This subsection discusses peoples performance in integrating information between (or across) sentences they have read. Developers need to integrated information from different source code statements, to obtain a

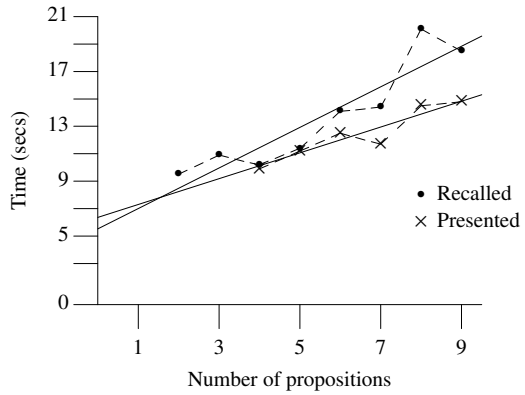


Figure 1707.3: Reading time (in seconds) and recall time for sentences containing different numbers of propositions (straight lines represent a least squares fit; for reading $t = 6.37 + .94P_{pres}$, and for recall $t = 5.53 + 1.48P_{rec}$). Adapted from Kintsch and Keenan.[735]

higher-level view of a programs behavior. It is assumed that the main factors affecting the performance of a reader of prose sentences are also the main factors, and in the same proportions, that affect the performance of readers of source code statements. The factors are working memory and the processing performed on the information it contains (see Just and Carpenter[697] for a capacity theory of comprehension).

The relative order in which a developer has to write many statements is dictated by dependencies between the operands they contain. However, there is often a degree of flexibility in the absolute order in which they occur. For instance, some developers initialize all locally declared objects at the start of a function, while others initialize them close to where they are used.

Although various theories of text comprehension have been proposed,[697, 734] it is not yet possible to give reliable answers to detailed questions. For instance, in the following three assignments, would moving the assignment to x after the assignment to y reduce the cognitive effort needed to comprehend the value of the expression assigned to z?

```
1  x = ex_1 + ex_2;            /* Could be reordered to follow assignment to y.   */
2  y = complicated_expression; /* Contains no dependencies on previous statement. */
3  z = y + ex_1;
```

Optimizing statement ordering, in those cases where some flexibility is available, to minimize reader cognitive effort when integrating information between statements requires that the relationships between all statements within a function be taken into account. For instance, ex_2 may also appear prior to the assignment to x and there may be a greater benefit to this assignment appearing close to this usage, rather than close to the assignment to z.

Because there is no method of measuring adherence, no guideline recommendation, dealing with statement ordering, is given here.

reading span
0 memory
developer

A study by Daneman and Carpenter[313] investigated the connection between various measures of subjects working memory span and their performance on a reading comprehension task. The two measures of working memory used were the *reading span* and *word span*. In the reading span test subjects have to read, out loud, sequences of sentences while remembering the last word of each sentence, which have to be recalled at the end of the sequence. The number of sentences in each sequence is increased until subjects are unable to successfully recall all the last words. The word span measure is a variant of the digit span test discussed elsewhere, using words rather than digits.

0 memory
digit span

In the Daneman and Carpenter study the reading comprehension task involved subjects reading a narrative passage containing approximately 140 words and then answering questions about facts and events described in the passage. Various passages were constructed, where the distance between information needed to answer

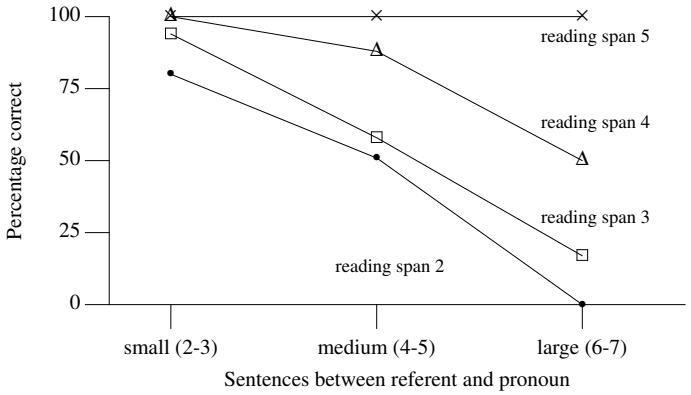


Figure 1707.4: Percentage of correct subject responses to the pronoun reference questions as a function of the number of sentences between the pronoun and the referent noun. Plotted lines are various subject reading spans. Adapted from Daneman and Carpenter.[313]

some of the questions was varied. For instance, the final sentence of the passage contained a pronoun (e.g., she, her, he, him, or it) that referred to a noun occurring in a previous sentence. Different passages contained the referenced noun in either the second, third, fourth, fifth, sixth, or seventh sentence before the last sentence.

In the excerpt: " . . . river clearing . . . The proceedings were delayed because the leopard had not shown up yet. There was much speculation as to the reason for this midnight alarm. Finally he arrived and the meeting could commence." the question "Who finally arrived?" refers to information contained in the last and third to last sentence. The question "Where was the meeting held?" requires the recall of a fact.

The results showed that there was little correlation between a subjects performance in the word span test and the reading comprehension test. However, there was a strong correlation between their performance in the reading span test and the reading comprehension test (see Figure 1707.4). A similar pattern of results was obtained when the task involved listening, rather than reading. A second experiment included controls to ensure that subjects processed the sentences in the reading span test (rather than simply looking for and remembering the last word) and that differences in rate of reading were not a factor. The pattern of results was unchanged. A study by Turner and Engle[1371] found that having subjects verify simple arithmetic identities, rather than a reading comprehension test, did not alter the results. However, altering the difficulty of the background task (e.g., using sentences that required more effort to comprehend) reduced performance.

The difference between word span and reading span as measures of working memory is that the first is purely a measure of memory usage while the second also involves processing information (the extent to which processing information consumes, interferes with, or competes with working memory resources is a hotly debated issue). Comprehension of prose involves integrating information within and between sentences. Other studies have found that reading span correlates with performance in a number other tasks, including:

- A study by Daneman and Carpenter[314] investigated subjects performance in integrating information within a single sentence and across two sentences. For instance, is there a difference in comprehension performance between the sentence "There is a sewer near our home who makes terrific suits" (this is an example of what is known as a *garden path* sentence) and "There is a sewer near our home. He makes terrific suits"? The results (see Figure 1707.5) show that a sentence boundary can affect comprehension performance. It was proposed that this difference in performance was caused by readers purging any verbatim information they held, in working memory, about a sentence on reaching its end. The availability of previously read words, in the single sentence case, making it easier to change the interpretation made on the basis of what has already been read.
- There are a number of structural components involved in the reading of prose. For instance, at the microstructure level words are formed from character sequences and syntactic processing of words

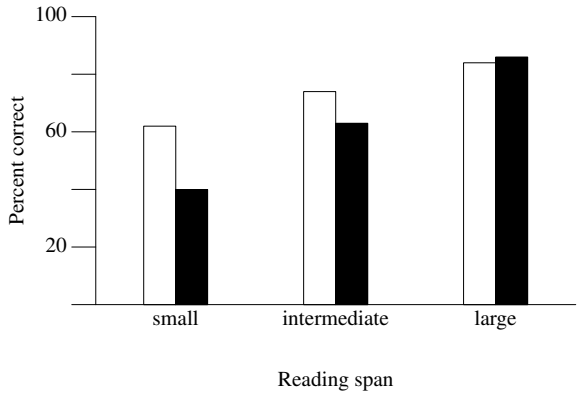


Figure 1707.5: Percentage of correct answers as a function of subject's reading span and the presence or absence of a sentence boundary. Adapted from Daneman and Carpenter.[314]

occurs, while at the macrostructure level information has to be integrated across sentences and the narrative has to be followed. A study by Graesser, Hoffman, and Clark[510] found that (1) more cognitive resources are allocated to macrostructure than microstructure processing, (2) differences in reading speed could be attributed to different rates of microstructure processing, and (3) variations in reader goals affect the rate at which the macrostructure is processed.

- A study by Glanzer, Fischer, and Dorfman[495] investigated the affects an interruption had on subjects performance, when reading prose. The results showed that when reading organized text (a sequence of sentences having dependencies between them) an interruption (requiring a different task to be performed) caused information on the last two sentences to be lost from working memory. This loss of information caused a reduced subjects performance in reading the remaining sentences (unless they were able to reread the previous two sentences). An interruption did not cause a reduction in performance when the sentences were not organized (i.e., they were not related to each other).
- A study by Gibson and Thomas[486] found that subjects were likely to perceive complex ungrammatical sentences as being grammatical. Subjects handling complex sentence that exceeded working memory capacity by *forgetting* parts of the syntactic structure of the sentence, resulting in a grammatically correct sentence.
- Text inference often involve more than integrating information between sentences. Knowledge about the real world is often required. A study by Singer and Ritchot[1245] showed subjects pairs of sentences and asked them to answer questions about inferences that could be drawn from these sentences. For instance, the two sentences "Valerie left early for the birthday party. She spent an hour shopping in the mall." would be expected to activate reader's knowledge of bringing birthday presents to parties, while the sentences "Valerie left the birthday party early. She spent an hour shopping in the mall." would not be expected to activate such knowledge. The results showed that there was no correlation between reading span and knowledge access when making such bridging inferences. The result of this study implies that while reading span provides a good measure of a person's ability to integrate information between sentences, it does not provide a measure of their ability to integrate information they have just read with information held in long-term memory.

A study by Ehrlich and Johnson-Laird[380] asked subjects to draw diagrams depicting the spatial layout of everyday items specified by a sequence of sentences. The sentences varied in the extent to which an item appearing as the object (or subject, or not at all) in one sentence appeared as the subject (or object, or not at all) in the immediately following sentence. For instance, there is referential continuity in the sentence sequence "The knife is in front of the pot. The pot is on the left of the glass. The glass is behind the dish.",

but not in the sequence "The knife is in front of the pot. The glass is behind the dish. The pot is on the left of the glass.".

The results found that when the items in the sentence sequences had referential continuity 57% of the diagrams were correct, compared to 33% when there was no continuity. Most of the errors for the non-continuity sentences were items missing from the diagram drawn and subjects reported finding it difficult to remember the items as well as the relation between them.

A study by Frase[446] found that the kind of deduction subjects had to perform (e.g., forward to backward chaining) also affects their performance. For instance, verifying the correctness of the deduction in "Some X are Y. Some Z are X. Therefore, some Z are Y." requires forward chaining, while verification of "Some Y are X. Some X are Z. Therefore, some Z are Y." requires backward chaining. The results showed that subjects were slower and more error prone when performing backward chaining.

statement visual layout

Visual layout of statements

Source code statements are generally written one per line, a practice that is different from prose sentences (which generally appear as a continuous stream of characters, with one sentence immediately following another on the same line, with line breaks decided by the space available for the next word). Statements are usually indented on the left to show block nesting level (this issue is discussed elsewhere).

reading 770 kinds of visual 792 skimming

```
1  #include <string.h>
2  #define v1 13
3  #define v2 0
4  #define v3 1
5  int v4(char v5[], int *v6) { int v7, v8; *v6=v2; v8=strlen(v5); if
6  (v8 > v1) { *v6=v3; } else { for (v7=0; v7 < v8; v7++) { if ((v5[v7]
7  < '0') || (v5[v7] > '9')) { *v6=v3; } } } }
```

The two cases discussed here are the practice of writing two or more statements on the same line and what to do when a statement will not fit on a single line. These cases are only issues that need discussion because source is not always read in detail, it is sometimes rapidly scanned visually. During rapid visual scanning of the source its left edge is often used as a reference point for the start of individual statements. This usage suggests the following:

reading 770 kinds of

- If multiple statements appears on the same line, readers using rapid visual scanning may only the first one may be noticed. For this reason some coding guideline documents recommend that a line contain at most one statement. However, there can be benefits to placing more than one statement on the same line. For instance, in the following fragment assigns values representing the four corners of a square:

  ```
  1  x1=1.0; y1=4.0; x2=3.0; y2=4.0;
  2  x3=1.0; y3=2.0; x4=3.0; y4=2.0;
  ```

  and the layout of the statements is suggestive of what they represent (other combinations are possible), and has the possibly benefits of reducing comprehension effort and being able to have more statements simultaneously visible on the screen. These benefits need to be balanced against the potential cost of readers failing to notice the assignment to other objects on the same line. Within a **`switch`** statement some of the labeled statements may only perform a single operation. It is possible to visually highlight this operation and contrast it with operations performed by other labeled statements by placing the *uninteresting* termination statement to the right of the statement, rather than underneath it.

  ```
  1  case 1: i--;            break;
  2  case 2: k++;            break;
  3  case 3: ++expr;ssion--; return;
  4  case 4: func();         break;
  ```

  These benefit need to be balanced against the potential cost of readers failing to notice a difference in termination statements, or treating two statement as one statement. The following guideline recommendation allows for some degree of flexibility.

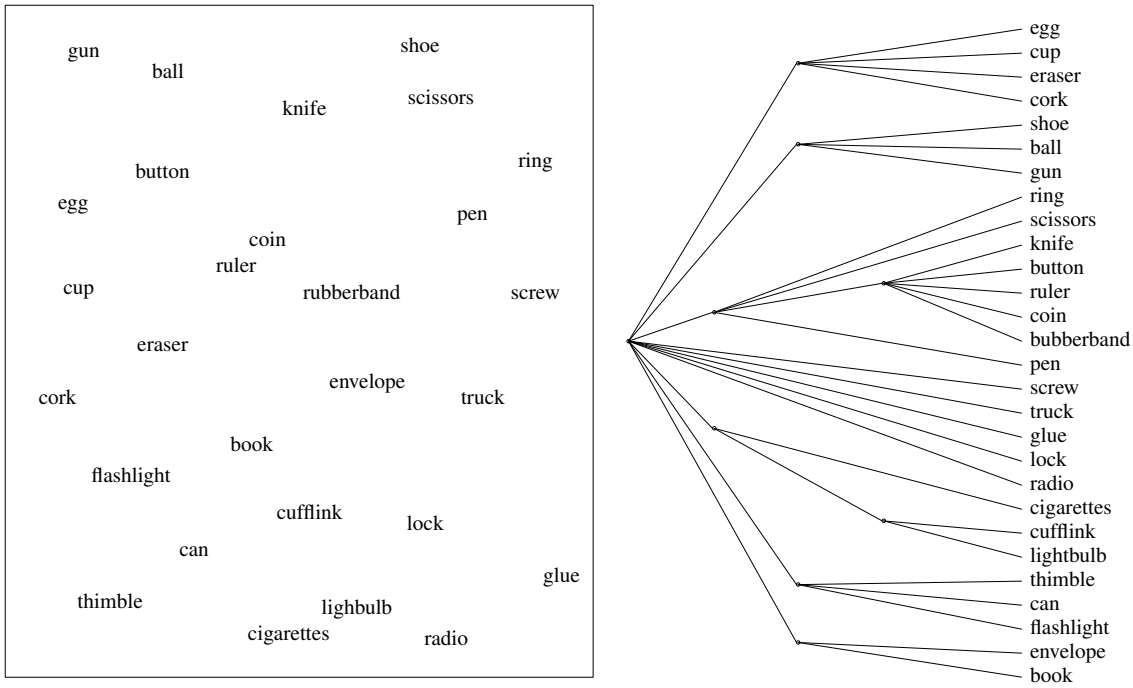


Figure 1707.6: Example of an object layout and the corresponding ordered tree for one of the subjects. Based on McNamara, Hardy, and Hirtle.[923]

Rev 1707.1

Multiple statements shall only appear on the same line, in a visible form, when the probable reduction in reader comprehension costs is greater than the probable increase in costs caused by mistakes made when quickly scanning the source visually.

- Readers are likely to treat the start of every line that appears immediately above or below adjacent lines, in a sequence of statements, as the start of a different statement. One way of reducing this possibility is to use a significant amount of right indentation on the second and subsequent lines relative to the start of the first line (which is what developers appear to do).

grouping
spatial location

Studies have found that peoples memory for objects within their visual field of view is organized according to the relative positions of those objects. For instance, a study by McNamara, Hardy, and Hirtle[923] gave subjects two minutes to memorize the location of objects on the floor of a room (see Figure 1707.6). The objects were then placed in a box and subjects were asked to place the objects in their original position. The memorize/recall cycle was repeated, using the same layout, until the subject could place all objects in their correct position.

The order in which each subject recalled the location of objects was used to create a hierarchical tree (one for each subject). The resulting trees (see Figure 1707.6 for an example) showed how subjects' spatial memory of the objects seen had a hierarchical organization, with the spatial distance between items being a significant factor in its structure.

Source code statements usually have a one-dimensional organization, down the display (indentation does not change the one-dimensional organization; multiple statements on the same line is not really fully two-dimensional).

If developer memory of statement sequences is hierarchical, with visual distance between statements being a significant factor in the formation of clusters, then ordering statements so that related ones are close to each

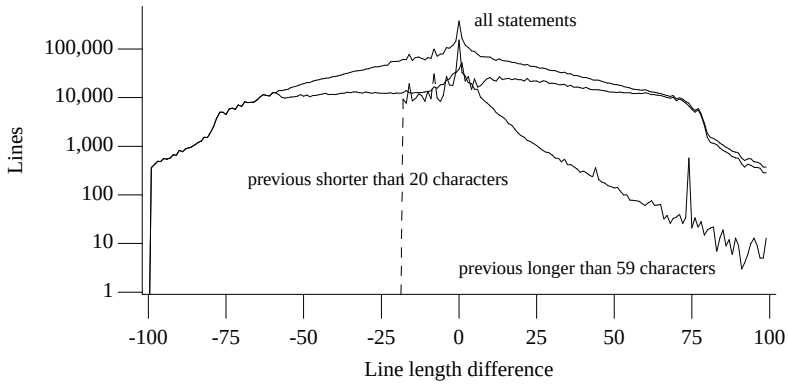


Figure 1707.7: Visible difference in offset of last non-space character on a line between successive lines, in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters), for lines of various lengths, i.e., those whose previous line contained 60 or more characters, and those whose previous line contains less than 20 characters. There are ten times fewer lines sharing the same right offset as sharing the same left offset (see Figure 1707.8). Based on the visible form of the `.c` files.

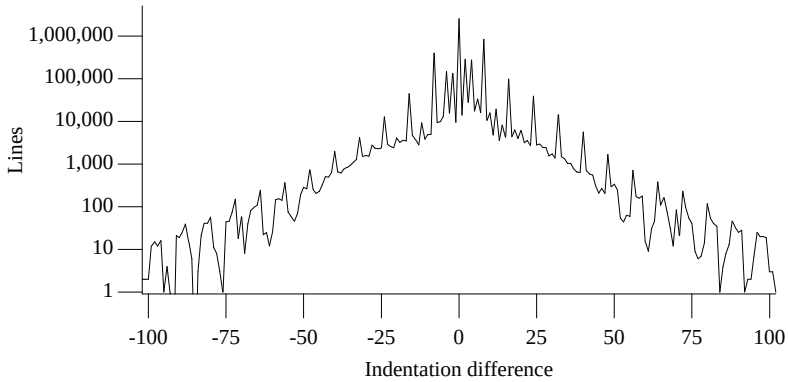


Figure 1707.8: Visible difference in relative indentation of first non-space character on a line between successive lines in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters). The smaller peaks around zero are indentation differences of two characters. The wider spaced peaks have a separation of eight characters. Individual files had more pronounced peaks. Based on the visible form of the `.c` files.

other may improve recall performance. The extent to which statements can be reordered will depend on the relative order in which their side effects must occur.

metrics introduction 0

Many software engineering metrics use statements as the unit of measurement.

Usage

Of the approximately 2,204,000 statements in the visible form of the `.c` files 60.3% were *`expression-statement`*s, 21.3% *`selection-statement`*s, 15.0% *`jump-statement`*s, and 3.4% *`iteration-statement`*s. Of these 5.4% were *`labeled-statement`*s.

Semantics

---

statement

A *`statement`* specifies an action to be performed. 1708

Commentary

This defines the term *statement*. This definition might also be said to include those declarations that include initializers. However, *`statement`* is also a terminal of the C syntax. Like declarations, statements can also cause identifiers to be declared (e.g., the type of a cast operator). So the two do share some characteristics. In the case of the null statement no action is performed.

null statement 1733

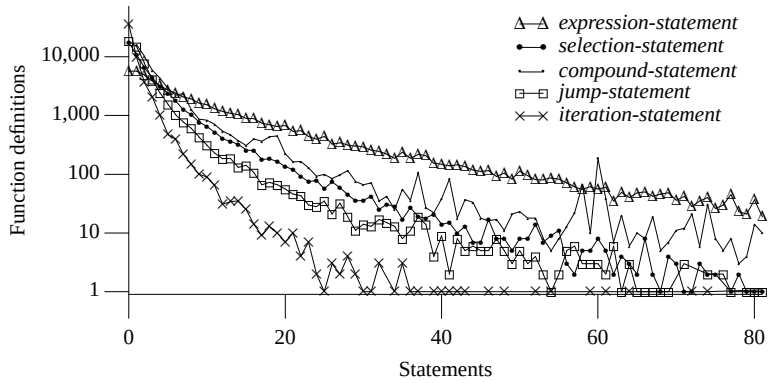


Figure 1707.9: Number of function definitions containing a given number of each kind of statement. Based on the translated form of this book's benchmark programs.

C++

The C++ Standard does not make this observation.

Other Languages

Some languages do not define things called *statement*s, everything is an expression. While other languages do define them, and always allow them to return values, just like C expressions. Programs written in logic programming languages (e.g., Prolog) contain a list of facts, not statements, to execute. The user of such programs states a goal and the implementation attempts to prove, or disprove it, using the available facts.

Coding Guidelines

It is possible to write statements whose action, when performed, does not affect the output of a program. This issue is discussed elsewhere. A sequence of one or more statements whose actions can never be performed is commonly known as *dead code*.

190 redundant code
190 dead code

1709 Except as indicated, statements are executed in sequence.

statement executed in sequence

Commentary

The sequence referred to here is that obtained by starting with the first token in a source file and parsing to the end of that file. The exceptions are caused by statements which change the flow of control, causing a statement at the start of another sequence to be executed. The terms *order of execution*, or *execution order* are sometimes used to describe the order in which statement sequences are executed. This specification has been interpreted as having a wider meaning by the C committee (see the response to DR #087). It is taken to imply that the execution of two functions cannot overlap (i.e., occur in parallel). For instance, in the expression `g() + h()` one of the functions is chosen to be executed and that function must execute a **`return`** statement before the other function can start execution.

Performing the actions specified by statements is what most programs spend most of their time doing (issues such as idling, waiting for an I/O to complete, and system resource management, such as page swapping, are not considered here). In some applications it is important to be able to make accurate estimates of the time needed to execute a given sequence of statements. For instance, in realtime applications there may be a fixed time window within which certain calculations must be carried out, either because new data is arriving or the result of the calculation is needed.

Accurately estimating the best and worst-case execution times (*BCET* and *WCET*) of a statement is not always as simple as summing the execution times of the sequence of machine instructions generated for that statement. Issues such as instruction pipelining, caching of data and instructions, and dependencies between instructions (that create interlocks) all complicate the calculation. Various tools have been developed for estimating the worst-case execution time of C source code. Engblom[391] reviews the available techniques

0 processor pipeline
0 cache

iteration statement 1763 syntax

and proposes worst-case execution time analysis method. The subproblem of estimating the number of iterations of a loop is discussed elsewhere.

Other Languages

Some languages (e.g., Algol 68) contain constructs that allow developers to specify that certain statements may be executed in parallel. Other languages (e.g., Java) support threads of execution where the function is the smallest unit of concurrent execution.

Common Implementations

Translators that perform little or no optimization usually generate machine code on a statement by statement basis. This severely restricts the savings that can be made and considering sequences of statements provides an opportunity to make greater savings (the actual unit considered is usually a basic block). The OpenMP C++ API specification[42] defines directives that can be included in source code to support symmetric multiprocessing (SMP).

basic block 1710

```
1  int f(void)
2  {
3  /* ... */
4     #pragma omp parallel
5     {
6     /*
7      * Code in this block potentially executed
8      * in another processor thread.
9      */
10    }
11 /* ... */
12 }
```

---

block

A *block* allows a set of declarations and statements to be grouped into one syntactic unit. 1710

Commentary

This defines the term *block*. The term *compound statement* and its association with block is discussed elsewhere. In various contexts a block might exist without there being a compound statement present.

compound statement 1729 syntax

block 1742 selection sub-statement

C90

A compound statement (also called a block) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in 6.1.2.4).

compound statement 1729 syntax

The term *block* includes compound statements and other constructs, that need not be enclosed in braces, in C99. The differences associated with these constructs is called out in the sentences in which they occur.

C++

The C++ Standard does not make this observation.

Common Implementations

basic block

The term *basic block* is often used by compiler writers to refer to a sequence of instructions that has one entry point and one exit point. The basic block is often the region of code over which many optimizers operate. More sophisticated optimizers consider the effects of multiple basic blocks. A block may also be a basic block, but if it contains any statements or operators that have conditional flow of control it will be a sequence of basic blocks. The implementation of C operators such as function call, logical-AND, and the comma operator all terminate one basic block and start another; as do the statements **`goto`** and **`return`** and labeled statements.

Table 1710.1: Occurrence of constructs that terminated execution of a basic block during execution of PostgreSQL processing the TPC-D benchmark. Adapted from Ramirez, Larriba-Pey, Navarro, Serrano, Torrellas, and Valero.[1141]

Basic Block Type	Static Count (thousand)	Dynamic Count (billion)
Branch	54.026 (42.4%)	4.0 (50.2%)
Fall-through	31.120 (24.4%)	1.8 (22.4%)
Function return	32.052 (25.2%)	1.1 (13.7%)
Function call	10.228 (8 %)	1.1 (13.7%)

Writers of translators and processor designers want the average number of machine instructions executed in a basic block to be as large as possible. For translators long sequences of code with continuous flow of control increases the probability of common subexpressions occurring and the consequent reuse, rather than recalculation, of values. For processors large basic blocks enable them to keep their pipelines full, minimizing average instruction execution time. Function inlining can remove one of the constructs that cause basic blocks to end, function calls (see Table 1710.1), increasing the length of some basic blocks. Table 1710.2 lists average basic block lengths for some complete programs. The special case of iteration statements is discussed elsewhere.

0 processor pipeline
1529 inline suggests fast calls
1774 for statement

Table 1710.2: Mean number of machine instructions executed per basic block (i.e., total number of instructions executed in a function divided by the total number of basic blocks executed in that function) for a variety of SPEC benchmark programs. *Leaf* refers to functions that do not call any other functions, while *Non-Leaf* refers to functions that contain calls to other functions. Based on Calder, Grunwald, and Zorn.[192]

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	6.8	4.9	eqntott	9.1	5.4
ditroff	6.8	4.7	espresso	5.0	5.1
tex	10.4	8.5	gcc	5.2	5.7
xfig	4.8	5.3	li	2.9	5.7
xtex	7.3	5.8	sc	3.5	4.2
compress	18.4	5.7	Mean	7.3	5.5

Just as optimizers moved up from working at the single statement level, they eventually moved up from working at the single basic block level (the issue of whole function optimization is discussed elsewhere). Translator output (machine code) for each block often occurs in the program image in the same sequential order as the basic blocks in the source code. However, processors can have a number of characteristics that can cause this ordering to be sub-optimal, including:

1821 function definition syntax

- *Span dependent branch instructions.* This issue is discussed elsewhere.
- *Instruction caches.* The size of a function or basic block, the total size of the cache and the size of a cache line all need to be considered when working out the optimal layout of instruction sequences in memory.[547]
- *Instruction pipelines*. Basic blocks can be reordered to ensure that they start on alignments best suited to a processor.[1495]
- *Storage organized in fixed sized pages.* Having infrequently executed instructions in the same page as frequently executed instructions can increase execution overheads (through increased swapping of pages). Storing infrequently executed instructions together in different pages can increase overall performance (the overhead of the extra instruction to jump back, at the end of an else arm, is offset by the removal of the instruction that would otherwise be needed, in the if arm, to jump around the else).

1783 jump statement causes jump to
0 cache
0 processor pipeline
1710 basic block
0 storage dividing up

The memory manager prefetches sequences of instructions to fill a complete cache line. Optimal use of this prefetching occurs if all instructions in a cache line are executed. In the example below, whichever arm of the **`if`** statement is executed, it is likely that some instructions from the non-executed arm will be fetched into a cache line.

```
if (x == 2)
   {
   /* Frequently executed code. */
   }
else
   {
   /* Infrequently executed code. */
   }
/* other code */
```

If it is known (for instance, through dynamic profiling information[986]) that one of the arms is executed more frequently than the other, it may be worthwhile reordering basic blocks in the program image. One technique is to place infrequently executed blocks in an area of storage reserved for such cases. The generated code being rewritten to jump to the infrequent block, and back again (the rewritten form of the above example is in C form below). This reorganization is likely to result in a higher percentage of the instructions contained in a cache line being executed.

```
if (x != 2)
   goto INFREQ_15_s;

   {
   /* Frequently executed code. */
   }
INFREQ_15_e:
/* other code */


/* ... */
goto INFREQ_14_e;
INFREQ_15_s:
   {
   /* Infrequently executed code. */
   }
goto INFREQ_15_e;
INFREQ_16_s:
/* ... */
```

The major problem associated with optimizing code layout at translation time is that information on frequency of basic block execution usage is needed. Developers may be unwilling or unable to gather this information, or it may be very difficult to obtain reliable information because of different end user usage patterns.

One solution to this problem is to perform what has been called *just in time code layout*.[220] The usage patterns of an executing program were monitored to decide when it is worthwhile moving code within the executing program image, to improve the instruction cache hit rate. Performance results comparable to those obtained for profile based methods have been obtained for programs running under Unix, but results for programs running under Windows NT were mixed (as they were for profile based code layout).

As processor power and available storage capacity has increased, the ability to consider code layout from more global perspectives has become possible. At first researchers built translators that reordered statements within basic blocks, and then moved on to reordering basic blocks within individual functions.[9] The break has now been made with source level constructs and the latest code layout algorithms are based purely on frequency of basic block execution over the entire program[539] (the function definitions to which the basic blocks might be said to belong have become irrelevant).

Usage

limit 277
block nesting

Usage information on block nesting is discussed elsewhere.

object initializer evaluated when

The initializers of objects that have automatic storage duration, and the variable length array declarators of 1711

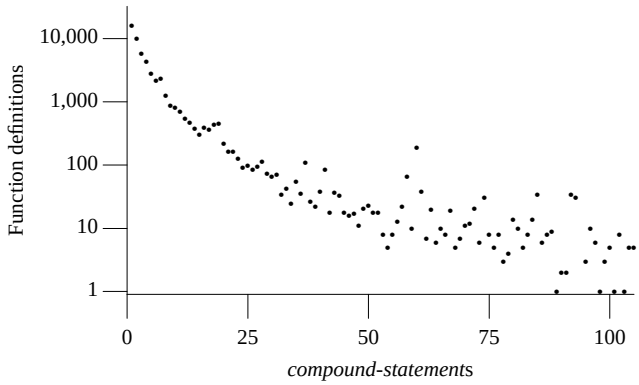


Figure 1710.1: Number of function definitions containing a given number of `compound-statement`s. Based on the translated form of this book's benchmark programs.

ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

Commentary

Both declarations and compound literals can create objects having automatic storage duration and an initializer. Storage for objects that have automatic storage duration, and non VLA type, will have been created at the start of their lifetime (when the block that declares them was entered). As well as performing initialization, the evaluation of a declaration for an object having a variable length array type also allocates storage for it.

457 automatic storage duration
464 VLA lifetime starts/ends

The indeterminate value stored in an object is a conceptual value in the sense that no pattern of bits need be stored in the object by an implementation (because no predictable behavior is required to occur, should this value be accessed).

75 indeterminate value

In one particular case, overlapping initializers, the evaluation of the initializers need not occur as if they were a sequence of assignment statements.

1676 initialization in list order

C90

Support for variable length array types is new in C99.

C++

Support for variable length array types is new in C99 and they are not specified in the C++ Standard.

Other Languages

This initialization behavior is found in the majority of programming languages (although many do not support the mixing of statements are declarations).

Common Implementations

The indeterminate value stored in an object may be affected by the initialization of other objects defined in the same block. For instance, if a several objects are initialized to the same value it may be more efficient to initialize a block of storage, which may include space allocated to uninitialized objects, using a loop rather than individual stores.

Coding Guidelines

The evaluation of variable length array declarators can generate side effects. However, because the expression appearing between **[** and **]** is not a full expression there is no sequence point after its evaluation (there is a sequence point at the end of the full declarator that contains it). For this reason the guideline given for the evaluation of full expressions is not applicable here. However, until the use of variable length arrays becomes more common and the ordering of side effects in their evaluation becomes an issue worth addressing a guideline recommendation is not cost effective.

1712 full expression
187 sequence points
1550 full declarator sequence point
187.1 sequence points all orderings give same value

```
#include <stdio.h>

int glob;

void f(void)
{
int
    /*
     * Unspecified order of evaluation of expressions in a full declarator.
     */
    a_1[printf("Hello ")][printf("world\n")],

    /*
     * Same object modified more than once between sequence
     * points -> undefined behavior.
     */
    a_2[glob++][glob++];
}
```

string literal 903
static storage duration

The extent to which developers might draw an, incorrect, parallel between a compound literal containing constant expressions only and string literals (which have static storage duration) is not known. One important difference is that any modification of the unnamed object, denoting the compound literal, will be overwritten when the original definition is executed again. Until more experience in developer use of compound literals has been gained there is no point in discussing this issue further.

Example

```
#include <stdio.h>

extern int glob;
struct T {
          int mem;
         };

void f(void)
{
int i = 0;

START_LOOP:
if (i == 10)
   goto END_LOOP;

i++;
   int      loc_1 = glob;
   struct T loc_2 = { 1 },             loc_3 = { glob },
           *ploc_a = (struct T){ 1 }, *ploc_b = (struct T){ glob };

            loc_1++;
            loc_2.mem++;               loc_3.mem++;
            ploc_a->mem++;             ploc_b->mem++;

goto START_LOOP;
END_LOOP:;

if (       (loc_1 != (glob+1))  ||
           (loc_2.mem != 1+1)   ||    (loc_3.mem != (glob+1)) ||
           (ploc_a->mem != 1+1) ||    (ploc_b->mem != (glob+1)))
   printf("This is not a conforming implementation\n");
}
```

1712 A *full expression* is an expression that is not part of another expression or of a declarator.

full expression

Commentary

281 parenthesized expression nesting levels

This defines the term *full expression*. While the term may only be used once outside of this C paragraph, it can be of use in disambiguating what is meant during conversations between developers. Developers often use the shorter term *expression*. However, this term might also be applied to individual arguments in a function call, or an array index. The terms *top-level expression* or *outer most expression* are sometimes used to refer to what the C Standard calls a full expression. A component of an expression is often referred to as a *subexpression*. An expression that is part of a declarator is included within the definition of a full declarator. 1549 full declarator

The subexpression `x+y` is said to be a *common subexpression* of `x+y + a[x+y]` (it may also be a CSE of other expressions, provided the values of `x` and `y` are unchanged).

common subexpression

C++

A full-expression is an expression that is not a subexpression of another expression. 1.9p12

The C++ Standard does not include declarators in the definition of a full expression. Source developed using a C++ translator may contain declarations whose behavior is undefined in C.

```
1  int i;
2
3  void f(void)
4  {
5  int a[i++][i++]; /* undefined behavior */
6                   // a sequence point between modifications to i
7  }
```

1713 Each of the following is a full expression:

Commentary

These are all of the contexts in which an expression, that needs to be evaluated during program execution, can occur.

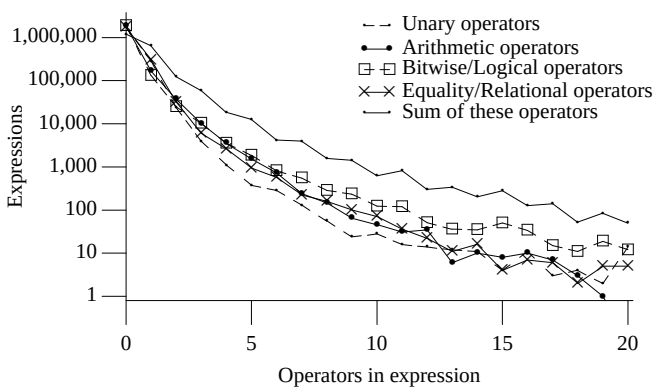


Figure 1712.1: Number of expressions containing a given number of various kinds of operator, plus a given number of all of these kinds of operators. Based on the visible form of the `.c` files.

C++

The C++ Standard does not enumerate the constructs that are full expressions.

Table 1713.1: Occurrence of full expressions in various contexts (as a percentage of all full expressions). Based on the translated form of this book's benchmark programs.

Context of Full Expression	Occurrence	Context of Full Expression	Occurrence
expression statement	65.9	**for** *expr-1*	1.6
if controlling expression	16.4	**for** controlling expression	1.5
return expression	6.2	**for** *clause-1*	1.5
object declaration initializer	4.2	**switch** controlling expression	0.6
while controlling expression	2.1		

---

full expression initializer

an initializer; 1714

Commentary

This is the complete expression appearing between braces, not the individual comma-separated expressions.

---

full expression expression statement

expression statement syntax 1731

the expression in an expression statement; 1715

Commentary

In an expression statement the entire statement is an expression.

C++

The following is not the same as saying that an expression statement is a full expression, but it shows the effect is the same:

6.2p1 *All side effects from an expression statement are completed before the next statement is executed.*

Other Languages

In many languages C's most common form of expression statement (i.e., simple assignment) consists of two expressions. These are the expressions appearing to the left and right of the assignment token, which are considered to be separate expressions.

---

full expression controlling expression

the controlling expression of a selection statement (**if** or **switch**); 1716

Commentary

This ensures consistent behavior for all control flows leading from the evaluation of the controlling expression. For instance, in the following sequence of statements:

```
1  if (i++)
2     j++;
3  k=i;
```

the value of `i` assigned to `k` does not depend on the sequence point contained within the arm of the **if** statement.

---

the controlling expression of a **while** or **do** statement; 1717

full expression controlling expression 1716

Commentary

The rationale here is the same as for selection statements.

---

each of the (optional) expressions of a **for** statement; 1718

Commentary

In C90 the semantics of the **for** statement were expressed in terms of an equivalent **while** statement. This equivalence required that each of the (optional) expressions be equivalent to a full expression.

1774 for statement

Other Languages

In many languages the expressions in a **for** statement are evaluated once, prior to the first iteration of the loop, every time the statement is encountered during program execution.

1719 the (optional) expression in a **return** statement.

return expression sequence point

Commentary

A sequence point occurs just before a function is called. Having one occur just before the called function returns allows the execution of a function call to be treated as an indivisible operation (from the point of view of the code containing the call).

1025 function call sequence point

1720 The end of a full expression is a sequence point.

full expression sequence point

Commentary

The order in which the sequence points at the end of full expressions occur is fully defined by a programs flow of control. Unlike the ordering of sequence points within a full expression (assuming there is more than one), which occur in one of several possible orderings.

187 sequence points

Coding Guidelines

The guideline recommendation dealing with sequence point evaluation ordering applies to sequence points within a full expression. The sequence point at the end of full expressions occurs in the same order for all implementations.

187.1 sequence points all orderings give same value

1721 **Forward references:** expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **return** statement (6.8.6.4).

6.8.1 Labeled statements

1722

labeled statements syntax

labeled-statement:
identifier **:** *statement*
case *constant-expression* **:** *statement*
default **:** *statement*

Commentary

Rationale

Case ranges of the form, *lo .. hi*, were seriously considered, but ultimately not adopted in the Standard on the grounds that it added no new capability, just a problematic coding convenience. The construct seems to promise more than it could be mandated to deliver:

- A great deal of code or jump table space might be generated for an innocent-looking case range such as `0 .. 65535`.
- The range `'A' .. 'Z'` would specify all the integers between the character code for "upper-case-A" and that for "upper-case-Z". In some common character sets this range would include non-alphabetic characters, and in others it might not include all the alphabetic characters, especially in non-English character sets.

No serious consideration was given to making switch more structured, as in Pascal, out of fear of invalidating working code.

When source code oriented symbolic debuggers (i.e., the ability to refer to lines in the source) are not available labels are sometimes used as a means of associating a symbol with a known point in the code.

Other Languages

Some languages (e.g., Fortran and Pascal) require that labels be digit, rather than alphabetic, character sequences.

Languages in the Pascal family do not classify **case**, or any equivalent to **default**, as a label. They are usually defined as part of the syntax of the selection statement in which they appear. Ada intentionally[619] differentiates visually between statement labels and case labels. An identifier that is a statement label is required to appear between the tokens << and >>. Some languages use **else** or **otherwise** rather than **default** in their selection statements.

Common Implementations

Some implementations (e.g., `gcc` and Code Warrior[927]) support case ranges as an extension.

Coding Guidelines

Many coding guideline documents recommend against the use of **goto** statements. However, in the case of C language, prohibiting labels (except **case** and **default**) would be more encompassing. For instance, a mistyped label in a **switch** statement may turn it into another kind of label (e.g., misspelling **default**, `defolt` is often seen for non-English speakers, creates a different kind of label). The issues associated with the use of jump statements are discussed elsewhere.

jump statement 1782 syntax

Usage

In the translated form of this book's benchmark programs 2% of labels were not the destination of any **goto** statement. Usage information on **goto** statements is given elsewhere.

jump statement 1783 causes jump to

Table 1722.1: Percentage of function definitions containing a given number of labeled statements (other than a **case** or **default** label). Based on the visible form of the `.c` files.

Labels	% Functions	Labels	% Functions
1	3.5	3	0.3
2	0.9	4	0.1

Constraints

case label appear within switch

A **case** or **default** label shall appear only in a **switch** statement. 1723

Commentary

This constraint is necessary because these constructs are not distinguished syntactically from other kinds of labels.

Other Languages

In other languages these kinds of labels are usually distinguished in the language syntax.

Coding Guidelines

This constraint does not prohibit **case** or **default** labels occurring in a variety of potentially surprising, to a reader, contexts within a **switch** statement. For instance, within a nested compound statement.

```
1  extern int glob;
2
3  void f(int valu)
4  {
5  switch (valu)
6     {
7     case 0: if (glob == 3)
```

```
8                   {
9                   case 2: value--;
10                  }
11              break;
12      }
13  }
```

In practice this usage is rare. Some practical applications of this usage are discussed elsewhere. 1766 Duff's Device

---

1724 Further constraints on such labels are discussed under the **switch** statement.

Commentary

This is essentially a forward reference. 66 forward reference

---

1725 Label names shall be unique within a function. label name unique

Commentary

Label names have function scope, which means they are visible anywhere within the function definition that contains them. 404 scope function

C90

This wording occurred in Clause 6.1.2.1 Scope of identifiers in the C90 Standard. As such a program that contained non unique labels exhibited undefined behavior.

A function definition containing a non-unique label name that was accepted by some C90 translator (a very rare situation) will cause a C99 translator to generate a diagnostic.

Other Languages

This requirement is common to most languages, although a few (e.g., Algol 68) give labels block scope. Some assembly languages supported duplicate label names. A jump instruction usually being defined, in these cases, to jump to the closest label with a given name.

Common Implementations

Although this was not a requirement that appeared within a Constraints clause in C90, all implementations known to your author issued a diagnostic if a label name was not unique within a function definition.

Coding Guidelines

Label names that are not referenced might be classified as redundant code. However, labels have uses other than as the destination of **goto** statements. They may also be used as breakpoints when stepping through code using a symbolic debugger. Automatically generated code may also contain labels that are not jumped to. Given that labels are relatively uncommon, and the only nontrivial cost involved in a redundant label name is likely to be a small increase in reader comprehension time, a recommendation that all defined labels be jumped to from somewhere does not appear to have a worthwhile benefit. 190 redundant code

The issues relating to the spelling of label names are discussed elsewhere. 792 label naming conventions

Semantics

1726 Any statement may be preceded by a prefix that declares an identifier as a label name.

Commentary

It is not possible to prefix a declaration with a label (although a preceding null statement could be labeled).

C does not provide any mechanism for declaring labels before they appear as a prefix on a statement. Labels may be referenced before they are declared. That is it is possible to **goto** a label that has not yet been seen within the current function.

Other Languages

Some languages (e.g., those in the Pascal family) provide declaration syntax for label names. Their appearance as a statement prefix is simply a use of a name that has been previously defined. Some languages (e.g., Fortran) allow all lines to be prefixed by a line number. These line numbers can also be used as the destination in **goto** statements.

---

case fall through

Labels in themselves do not alter the flow of control, which continues unimpeded across them. 1727

Commentary

EXAMPLE 1762 case fall through

An example of this is given elsewhere.

C++

The C++ Standard only explicitly states this behavior for **case** and **default** labels (6.4.2p6). It does not specify any alternative semantics for 'ordinary' labels.

Other Languages

In some languages certain kinds of labels alter the flow of control. For instance, in Pascal encountering a **case** label cases flow of control to jump to the end of the **switch** statement (the following using Pascal syntax and keywords).

```
1  case expr of
2     1: begin
3        x:=88;
4        y:=99;
5        end;
6
7     2: z:=77;
8     end;
```

Common Implementations

Labels may not alter the flow of control but they are usually the destination of a change of flow of control. As such they are bad news for code optimization, because accumulated information on the values of objects either has to be reset to nothing (worst-case assumption made by a simple optimizer), or merged with information obtained along other paths (more sophisticated optimizer).

Coding Guidelines

Most C coding guideline documents recommend against *falling through* to a statement prefixed by a **case** or **default** label. For instance, in the following fragment it is likely that the author had forgotten to place a **break** statement after the assignment to y.

```
1  case 1: x=88;
2          y=99;
3
4  case 2: z=77;
```

Duff's Device 1766

There are occasions where falling through to a statement prefixed by a **case** label is the intended behavior. A convention sometimes adopted is to place a comment containing the words FALL THROUGH on the line before the statement that is fallen into (some static analysis tools recognize this usage and don't issue a diagnostic for the fall through that it comments).

Experience suggests that code containing a statement prefixed by a **case** label whose flow of control *falls through* to another statement prefixed by a **case** label causes readers to spend significant effort in verifying whether the usage is intended or a fault in the code. Commenting such usage has a cost that is likely to be significantly smaller than the benefit.

Cg 1727.1

If the flow of control can reach a statement prefixed by one or more **case** labels, other than by a direct jump from the controlling expression of a **switch** statement, the source line immediately before that statement shall contain the words "FALL THROUGH" in a comment.

Table 1727.1: Common token pairs involving a **case** or **default** label. Based on the visible form of the `.c` files. Almost all of the sequences { case occur immediately after the controlling expression of the **switch** statement.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token
; default	0.4	81.4
; case	2.1	52.1
: case	15.5	22.1
{ case	2.6	15.0
} case	1.3	7.3
: default	0.5	5.7
#endif default	0.8	4.4

1728 **Forward references:** the **goto** statement (6.8.6.1), the **switch** statement (6.8.4.2).

6.8.2 Compound statement

compound statement syntax

1729

compound-statement:
{ *block-item-list*$_{opt}$ **}**
block-item-list:
block-item
block-item-list block-item
block-item:
declaration
statement

Commentary

Compound statements are commonly used to group together a sequence of statements that are required to be executed under the same set of conditions. While compound statements also provide a mechanism to limit the scope of declared identifiers and the lifetime of objects, developers do not often make use of this functionality (see Figure 408.1).

C90

The C90 syntax required that declarations occur before statements and the two not be intermixed.

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ **}**
declaration-list:
declaration
declaration-list declaration
statement-list:
statement
statement-list statement

C++

The C++ Standard uses the C90 syntax. However, the interpretation is the same as the C99 syntax because C++ classifies a declaration as a statement.

Other Languages

Many languages use the keywords **begin/end** instead of a pair of braces. While Fortran, prior to the 1995 standard, did not provide any syntactic mechanism for grouping statements together in blocks. Subroutines were the only explicit statement grouping construct. Developers used conditional and unconditional jumps to create groups of statements.

Some languages (e.g., Pascal) do not allow declarations to occur within nested blocks.

In some languages (e.g., Pascal) the semicolon is a statement separator, not a terminator as in other languages (e.g., Ada). So it need not occur after the last statement in a compound statement. But it may need to occur after the compound statement.

```
1  A := B;
2  begin C := D end;
3  E := F
```

In C the semicolon is part of the syntax of certain statements. The syntax for compound statement does not include a semicolon.

```
1  a = b;
2  { c = d; }
3  e = f;
```

Ada supports the optional occurrence of a token between the **end** and **;**, echoing the corresponding opening header. For instance, `end if;` terminating an **if** statement.

Common Implementations

A compound statement could be translated into a sequence of machine code instructions making up a basic block, provided it contains no statements or operators that cause conditional execution to occur. For instance, an **if** statement, or the logical-AND operator.

Coding Guidelines

The issues of mixing declarations and statements, and whether to locate declarations at the start of a block or near the point of first use are discussed elsewhere.

mixing 1821 declarations and statements

Many of the visual layout schemes for source code can be differentiated by where they place opening brace, relative to other tokens. A more realistic analysis, compared to the glowing claims of readability made by the proponents authors of these schemes, is given elsewhere. Some layout schemes locate the open brace after non-white-space characters (e.g., `if (x < y) {`), rather than on a line by itself (and possibly other white-space characters). There are several reasons for believing that the former usage results in a larger number of mistakes being made than the latter. These include: (1) the **{** token not being visible during a visual scan of the left edge of the visible source, and (2) non-white-space characters appear along the visual line connecting the opening and closing brace characters (which require cognitive effort to search and are a possible cause of interference).

reading 770 practice

reading 770 kinds of

Given the lack of experimental measurements of reading performance for different layout schemes, there is no empirical evidence to support any guideline recommendation relating to the visual layout of blocks.

Usage

Usage information on the number of declarations occurring in nested blocks is given elsewhere (see Figure 408.1).

Semantics

---

compound statement is a block

A *compound statement* is a block. 1730

Commentary

block 1710

This defines the term *compound statement*.

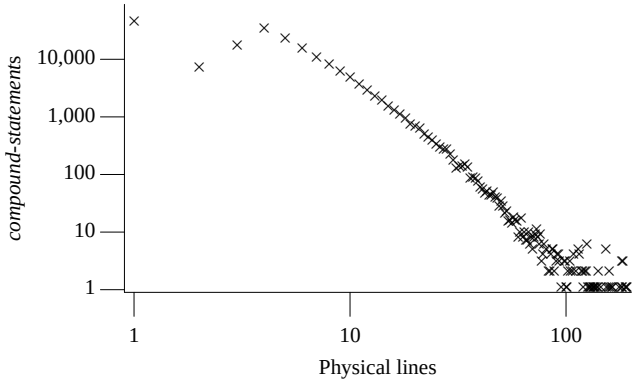


Figure 1729.1: Number of *compound-statement*s containing the given number of physical lines (including the opening and closing braces and any nested *compound-statement*s, but excluding the lines between the braces denoting the start/end of the function definition). Based on the translated form of this book's benchmark programs.

C90

The terms *compound statement* and *block* were interchangeable in the C90 Standard.

Coding Guidelines

The issue associated with enclosing the bodies associated with selection and iteration statements in a pair of matching braces is discussed in the C sentences for those statements. The term most commonly used by developers is *block*, rather than *compound statement*. In C99, but not C90, the term *block* includes constructs that are not compound statements. Authors of coding guideline documents need to ensure that their usage of terminology is consistent.

6.8.3 Expression and null statements

1731 null statement syntax expression statement syntax

expression-statement:
expression$_{opt}$ **;**

Commentary

Syntactically a sequence of two semicolons (e.g., `;;`) represents at least one null statement (and never a null declaration).

Other Languages

The syntax of many languages do not allow expressions to occur at the statement level. Assignment is often specified by placing an expression on either side of a token (such as `:=` or `=`) that is not defined to be an operator. Some languages (e.g., Fortran and PL/1) require that functions calls at the statement level be prefixed by the keyword **`call`**, while others (e.g., Ada and Pascal) support such calls (to *procedures*) without the use of any keyword. Perl supports what are known as statement modifiers, which provide a conditional or looping mechanism. For instance, `$i=$num if ($num < 50);` only performs the assignment if the controlling expression of the **`if`** is true (it is also possible to use **`unless`**, **`while`**, and **`until`** instead of **`if`**).

Table 1731.1: Occurrence of the most common forms of expression statement (as a percentage of all expression statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Form of *expression-statement*	%	Form of *expression-statement*	%
function-call	37	object = expression	4
object = object	16	object `v++`	2
object = function-call	10	expression	1
object = constant	7	other-expr-stmt	22

Semantics

expression statement evaluated as void expression

The expression in an expression statement is evaluated as a void expression for its side effects.[132)] 1732

Commentary

Treating an expression as a statement simplifies the C syntax. However, this specification is needed to handle the resulting value.

Other Languages

compound expression 1313

Some languages (e.g., Algol 68 and as an extension in `gcc`) allow some form of bracketed statement sequence to appear as the operand in an expression. In such languages the equivalent form of a C expression statement can return a value.

Coding Guidelines

EXAMPLE discard function return 1734

Some coding guideline documents recommend that function calls whose return value is not used (i.e., having a return type other than **`void`**) be explicitly cast to **`void`**. The rationale is that the cast provides an explicit indication of the authors intent to discard the functions return value (the return value presumably being used in other situations, otherwise a return type of **`void`** would be applicable). One of the costs of explicitly specifying that a functions return value is discarded is an increase in the cognitive effort needed to locate the name of the called function (which will appear between some tokens, rather than as the first token on a line). Given that the benefits do not appear to be significantly greater than the costs, no guideline recommendation is given here.

redundant code 190

The issue of expressions whose evaluation does not cause side effects is discussed elsewhere.

null statement

A *null statement* (consisting of just a semicolon) performs no operations. 1733

Commentary

This defines the term *null statement*. A null statement performs no operation in the sense that translators are unlikely to generate any machine code for it. It may perform a syntactic operation by allowing a label to be placed anywhere in the source.

EXAMPLE labeled null statement 1737

Other Languages

Some form of statement that performs no operation is common to most languages.

Coding Guidelines

Null statements are very easy to accidentally create and often go unnoticed by readers. Highlighting those cases where a null statement is intended can reduce a reader's effort deducing this fact (sometimes a semicolon as the only non-white-space character on a line is considered to be sufficient highlighting). Those cases where a null statement is not intended are faults and it is not the purpose of these coding guidelines to recommend against faults.

EXAMPLE null statement 1736

guidelines not faults 0

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  if (glob == 3);
6     ; /* A null statement that might be noticed while scanning the left edge. */
7
8  if (glob == 3);
9     /* A null statement unlikely to be noticed while scanning the left edge. */ ;
10
11 while (glob++ < 22);
12   ;  ;  ;  ;  ;  ; /* Conspicuous consumption of attention. */
13 }
```

1734 EXAMPLE 1 If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

EXAMPLE discard function return

```
int p(int);
/* ... */
(void)p(0);
```

Other Languages

Some languages contain functions and procedures, the former always returns a value, while the latter never returns a value and is required to be used in expression statement contexts.

Coding Guidelines

The issues associated with this usage are discussed elsewhere.

1732 expression statement evaluated as void expression

1735 132) Such as assignments, and function calls which have side effects.

footnote 132

Commentary

Or accessing an object declared with the **volatile** type qualifier.

C++

The C++ Standard does not contain this observation.

1736 EXAMPLE 2 In the program fragment

EXAMPLE null statement

```
char *s;
/* ... */
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

Coding Guidelines

The issues associated with side affects appearing in a controlling expression are discussed elsewhere.

1740 controlling expression if statement

1737 EXAMPLE 3 A null statement may also be used to carry a label just before the closing } of a compound statement.

EXAMPLE labeled null statement

```
while (loop1) {
        /* ... */
        while (loop2) {
                /* ... */
                if (want_out)
                        goto end_loop1;
```

```
                /* ... */
        }
        /* ... */
end_loop1: ;
}
```

Commentary

A null statement may also be used to carry a label just after the opening **{** of a compound statement (which may contain declarations with initializers).

Forward references: iteration statements (6.8.5). 1738

6.8.4 Selection statements

---

selec-
tion statement
syntax 1739

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
switch (*expression*) *statement*

Commentary

This syntax is ambiguous in that it does not uniquely specify the parse for the token sequence `if (a) b; if (c) d; else e;`. This syntactic ambiguity is resolved by a semantic rule.

else 1747
binds to nearest if

Other Languages

There is a great deal of variety in the syntactic forms of selection statements used by different languages. Some languages (e.g., those in the Pascal family) require that the controlling expression be followed by the keyword **then**, while others require the **if** statement to be terminated by some keyword (e.g., **end** in Ada, or **fi** in Algol 68). Some languages support an **elif** form (e.g., Ada, Algol 68, and the C preprocessor). Fortran has what it calls an *arithmetic if* statement:

preprocessor 1854
directives
syntax

```
1       if (i) 10, 20, 30
2      C Jump to label 10 if i < 0
3      C Jump to label 20 if i == 0
4      C Jump to label 30 if i > 0
```

The Fortran equivalent of the **switch** statement is known as a *computed goto* (in the following code fragment the **goto** jumps to label 10 if I==1, 20 if I==2, and so on):

```
1       goto (10, 20, 30, 40), I
```

Algol 60 required the labels to be declared in a **switch** statement:

```
1  SWITCH x := label1, label2, label3;
2        ...
3  GOTO x[i];
```

The Algol 68 **case** statement essentially has Fortran semantics (it also supports a **default** form), but with a Pascal like syntax.

Most languages (including Java) unconditionally bracket, syntactically, the complete list of case labels and their associated statements that follow a **switch** header.

Java supports the occurrence of labels that do not label any statement, at the end of the sequence of statements in a **switch** statement (where they essentially act as a comment whose contents are syntactically and semantically checked).

Languages in the Pascal family use the keyword **case**, rather than **switch**, is used. The following example is a fragment of Ada:

```
case x is
   when 2 => y:=1;
   when 3 => y:=2;
end case;
```

A few languages supports a **switch** statement containing more than one dimension (e.g., CHILL supports an arbitrary number, while Cobol supports a maximum of two). The following example is a fragment of Cobol:

```
        EVALUATE price ALSO quantity
           WHEN ANY ALSO 0   PERFORM ABC
           WHEN 0   ALSO ANY PERFORM PQR
           WHEN 1   ALSO num PERFORM XYZ
        END-EVALUATE
```

Common Implementations

The availability of the **switch** statement does not mean that developers always use it. Uh[1385] gives the following example from the source of `grep`.

```
if ((c = *sp++) == 0)
   goto cerror;
if (c == '<') { ... }
if (c == '>') { ... }
if (c == '[') { ... }
if (c == ']') { ... }
if (c >= '1' && c <= '9') { ... }
```

Uh added optimizations to an existing compiler to detect and optimize linear sequences of **if** statements that compared the same variable against different constant values. Performance improvements varied between 0.67% to 5.56%, depending on the processor (see Table 1753.1).

Yang, Uh, and Whalley[1490] took a different approach to optimizing nested sequences of **if** statements. They used profile information to reorder the sequence in which the controlling expression tests were made, putting those that succeeded most often first. An average performance improvement of 4% was obtained (on a variety of Unix tools).

The implementation of the **if** statement (and some operators, e.g., logical-AND) usually uses a conditional branch machine instruction. Conditional branch instructions create a bottleneck for processors that have the ability to execute more than one instruction at the same time. Without knowing which sequence of instructions are going to be executed, after the branch instruction, the processor is unable to keep the execution pipeline full; execution stalls. Several methods have been used to help reduce the likelihood of stalls occurring, including:

1248 logical-AND-expression syntax

- *Speculative execution of the two flows of control.* Speculative execution of two flows of control is implemented by keeping the results of both execution paths in shadow registers. Once the result of the branch test is known the values in the appropriate set of shadow registers are copied into the actual registers. This approach requires a lot of hardware resources (it has been implemented on the IBM 360/91 and the Sun SuperSPARC[1498]).
- *Predicting the branch the instruction is likely to take*, so called *branch prediction*, and speculatively executing that flow of control. If the prediction turns out to be correct the processor has a full pipeline of instructions and can continue from the point it had speculatively executed to, otherwise the pipeline has to be flushed and filled with instructions from the start location of the other branch.
- *Conditional instructions.* An alternative to a conditional branch is to use instructions whose execution is conditional on the setting of various processor flags. These instructions are read from the instruction stream but are only executed if the condition specified, as part of the encoding of the instruction, is currently true. The current conditions being set by executing a compare instruction, much like that which might appear before a conditional branch instruction.

1739 conditional instructions

branch prediction

Branch prediction

jump statement syntax 1782

Most modern high-performance processors perform some kind of branch prediction followed by speculative execution of the predicted destination instructions (unconditional branches are discussed elsewhere). Processor based branch prediction can take the following two forms:

- Encoding the branch instruction to include a bit specifying whether the branch is likely to be taken or not (e.g., the IBM PowerPC). This branch probability decision bit is filled in by the translator when generating machine code.
- Using the history of an executing programs previous branch decisions (taken/not taken) is used to predict the probably outcome of the next branch decision. Studies have found that there is often a strong correlation between the branch decisions made by a particular instruction (i.e., the same decision is repeated), also the last branch decision (usually a completely different branch instruction) and the decision made by the following branch instruction may be correlated (some branch predictors maintain information on the history of the branch itself and the branch previous to it, in the dynamic execution flow). This branch history information is often held in a table (a few bits specifying the direction of the last few jump decisions), indexed by the branch instructions address (usually a few of the least significant address bits).

The following are two methods a translator can use to predict branch direction:

- *Static analysis of the translated source code* (i.e., the machine code making up the program image). Various heuristics have proven to be reliable indicators of branch direction. This technique is known as *program-based* branch prediction.
- *Gather data on the branch decisions made during program execution.* This profile of branch decisions is then used in a subsequent translation to build a program image whose branch instructions are encoded to specify the most frequent branch direction taken during the profiling executions. This technique is known as *profile-based* branch prediction.

Static analysis of branch direction

The quantity measured in branch prediction is the *miss rate* (predicted destination incorrect), expressed as a percentage. The *perfect* predictor selects the branch direction based on the most common direction actually chosen, for each branch, over a large number of executions of the program. A study by Ball and Larus[88] analyzed translator generated machine code to create a number of heuristics that could be used to predict branch behavior. The heuristics found to be worthwhile (see Table 1739.1) were:

- *Opcode heuristic.* A relational comparison against zero is assumed to fail if it tests for less than (or less than or equal) and assumed to succeed if it tests for greater than (or greater than or equal).
- *Loop heuristic.* The successor does not postdominate[1739.1] the branch and is either a loop head or a loop preheader (i.e., passes control unconditionally to a loop head which it dominates). If the heuristic applies, predict the successor with the property.
- *Call heuristic.* The successor block contains a call or unconditionally passes control to a block with a call that it dominates, and the successor block does not postdominate the branch (many conditional calls handle exceptional cases, which are rarely taken). If the heuristic applies, predict the successor without the property.
- *Return heuristic.* The successor block contains a return or unconditionally passes control to a block that contains a return. If the heuristic applies, predict the successor without the property.

[1739.1] A node y, of some flow graph, is said to *postdominate* the node x if every path from x to the end of the graph includes y.

- *Guard heuristic.* Register r is an operand of the branch instruction, register r is used in the successor block before it is defined, and the successor block does not postdominate the branch (this test can only be performed after code generation and assumes that a translator has performed some form of optimizing register assignment). If the heuristic applies, predict the successor with the property.
- *Store heuristic.* The successor block contains a store instruction and does not postdominate the branch. If the heuristic applies, predict the successor without the property,
- *Pointer heuristic.* A test for equality between two pointers (one of them possibly being the null pointer constant) is assumed to fail. A test for inequality between two pointers (one of them possibly being the null pointer constant) is assumed to succeed.

Table 1739.1: Dynamic breakdown of non-loop branches for programs in SPEC89. *% of All Branches* is the percentage of all branches that are non-loop branches. *Heuristics* are the results of using the heuristics for predicting the target successor of each non-loop branch, *Perfect* the results for the perfect predictor, *Random* the results for predicting each non-loop branch randomly. *Big* is the number of non-loop branches in the program contributing more than 5% of all dynamic non-loop branches (and in parenthesis as a percentage of non-loop branches). Based on Ball and Larus.[88]

Program	% of All Branches	Heuristics	Perfect	Random	Big (%)	Program	% of All Branches	Heuristics	Perfect	Random	Big (%)
gcc	73	37	11	50	0 (0)	poly	20	40	3	31	3 (54)
lcc	71	32	12	52	1 (13)	fpppp	86	42	9	41	0 (0)
qpt	70	26	9	52	0 (0)	costScale	71	29	21	49	6 (52)
compress	66	40	18	66	6 (69)	doduc	52	33	3	49	0 (0)
xlisp	62	28	7	50	0 (0)	tomcatv	38	2	0	50	2 (98)
addalg	52	43	30	43	7 (67)	dcg	21	15	4	46	4 (51)
ghostview	52	16	4	47	4 (53)	spice2g6	21	36	8	52	2 (27)
eqntott	49	50	25	50	2 (92)	sgefat	18	26	8	61	8 (73)
rn	48	34	1	51	3 (25)	dnasa7	10	32	4	55	4 (58)
grep	44	1	0	3	3 (96)	matrix300	4	33	0	66	3 (99)
congress	40	28	3	57	2 (10)	Mean		29	10	49	
espresso	37	26	13	42	3 (24)	Std.Dev.		12	8	13	
awk	29	14	3	57	4 (29)						

A more detailed analysis of this heuristic approach can be found in Deitrich.[342] An approach that looked at source code was investigated by Sokolova,[1264] who reported some improvements. Calder[190] trained a neural network, using a corpus of programs, to infer the branching behavior of new programs (they called the approach *evidence-based* static prediction).

Conditional instructions

conditional instructions

The extent to which large sophisticated branch prediction circuitry can be incorporated into processors used in resource limited environments is limited by the significant amount of power it consumes (up to 10% of a processors total dynamic power dissipation,[1053] which drains batteries and heats up the device). An alternative solution to the bottleneck created by conditional branches is to do away with them. Some processors (e.g., ARM[56] and Intel Itanium[231,841,1342]) achieve this by making the execution of all instructions conditional. This is implemented by having a sequence of bits in the instruction encoding represent various conditions. During program execution fetched instructions are only executed if the condition encoded in their bit sequence is true. Conditional execution of instructions can be more efficient than using a conditional branch instruction when there are only a few *nulled* instructions (representing the unexecuted arm). Since the number of instructions in the arms of **if** statements is often small, use of such instructions can often be worthwhile ([231] gives empirical results).

The following discussion and example is based on one appearing in the Intel XScale Microarchitecture Programmers Reference Manual.[629]

```
1  int f(int x)
2  {
```

```
3  if (x > 10)    /* if-cond   */
4     return 0;   /* if-stmt   */
5  else
6     return 1;   /* else-stmt */
7  }
```

The unoptimized machine code generated, using branch instructions, for the **if** statement usually has the form:

```
    cmp  r0, 10
    ble  L1
    mov  r0,  0
    b    L2
L1: mov  r0,  1
L2:
```

while that generated using conditional instructions has the form:

```
    cmp   r0, 10
    movgt r0,  0
    movle r0,  1
```

Intel recommend[629] using the conditional instruction form when:

$$N1_C \times \frac{P1}{100} + N2_C \times \frac{100 - P1}{100} \leq N1_B \times \frac{P1}{100} + N2_B \frac{100 - P1}{100} + \frac{P2}{100} \times 4 \tag{1739.1}$$

where:

$N1_B$: Number of cycles to execute the *if_stmt* assuming the use of branch instructions.

$N2_B$: Number of cycles to execute the *else_stmt* assuming the use of branch instructions.

$P1$: Percentage of times the *if_stmt* is likely to be executed.

$P2$: Percentage of times a branch misprediction penalty is likely to be incurred.

$N1_C$: Number of cycles to execute the *if-else* portion using conditional instructions assuming *if-cond* to be true.

$N2_C$: Number of cycles to execute the *if-else* portion using conditional instructions assuming the *if-cond* to be false.

The code generated above requires three cycles to execute the *else-stmt* and four cycles for the *if-stmt*, assuming best-case conditions and no branch misprediction penalties.

Coding Guidelines

1 Introduction

conditional statement

Source code comprehension that involves a controlling expression of an **if** statement can have many facets. Some of these include the following:

- *Comprehending the conditional expression itself.* In particular the conditions under which it is true and false. These issues are discussed in detail in this subclause.
- *Interpreting the conditional expression within the application domain.* A conditional expression is used to make one of two choices and these may be driven by applications requirements (e.g., `line_len < MAX_LINE_LEN` might be the implementation of a requirement on the maximum number of characters that can appear on an output line). These application mapping comprehension issues are outside the scope of these coding guideline subsections and are not discussed further here.

- *Control flow dependencies.* An **if** statement may be nested within other **if** statements. The conditions represented by the additional controlling expressions may need to be taken into account,
- *Data dependencies.* When reading statements whose execution is directly affected by the flow of control of the conditional expression (e.g., an assignment statement in one of the arms of an **if** statement). These statements may need to be processed in the context of the conditions interpretation with the application or may involve operands that appear in the condition. For instance, in:

```
1  if ((x >= y) && (x <= z))
2     {
3     p++;
4     q=z-y;
5     ...
```

the reason for p being incremented is likely to be connected to what the condition represents within the application domain, while some information about the assignment to q can be obtained directly from the condition expression (e.g., it is always positive). A data dependency can also exist between an **if** statement and another such statement executed before it.

```
1  if (x == 2)
2     y = 45;
3
4  if (y != 45)
5     {
6     /* x does not equal 2 here */
7     }
```

Some coding guideline documents recommend a value for the maximum nesting of **if** statements (sometimes giving a rationale based on the limits of human short-term memory). Human short term memory does place constraints on the complexity of code than can be easily comprehended. However, while the time taken to comprehend a sequence of nested **if** statements may not feel like very long, it is much longer than the period of time over which short-term memory operates (a guideline recommendation based purely on human short-term memory capacity would probably need to limit nesting to a single level).

0 memory developer
0 Miller 7±2

Any guideline recommendation designed to deal with human cognitive capacity limits needs a reasonably accurate model of how the people involved are likely to process a given construct. Given that the controlling expressions of **if** statements can have significantly different complexities and couplings to other statements it is necessary to calculate how readers are likely to interpret and chunk the information present in them (i.e., simply counting nesting is much too simplistic). At the time of this writing it is not possible to perform this calculation. Like all guideline recommendation it is also necessary to consider what alternative constructs developers might use. In the case of **if** statements some of the issues include the following:

- Many *alternative* techniques don't remove the nesting, they simply rearrange the source. For instance, replacing the inner most nested **if** statements by a call to a function that contains them only reduces nesting depth because of the accounting practices used to measure depth (the measure is usually based on the contents of a single function; the contents of called functions are not examined).
- The implicit **if** statements implied by the use of the logical operators in a conditional expression. For instance, `if ((a < 1) && (b > 10))` is usually counted as a single **if** statement, while the equivalent form `if (a < 1) if (b > 10)` is counted as two (there have been few psychological studies comparing how people treat rephrasing of these two forms; one study[1163] found no psychological equivalence).

The rest of the discussion in this coding guideline subsection deals with the reader comprehension aspects of the dependencies between a controlling expression and the rest of the source code that contains it (other dependency issues are discussed elsewhere; while there has been a lot of talk about the software engineering

1821 function definition syntax

costs associated with dependencies there has been little experimental research on the topic). In particular it discusses studies of human performance in reasoning tasks. It is possible to draw many parallels between solving these tasks and comprehending the use of, and references to, conditional expressions. These coding guidelines make the assumption that the pattern of mistakes made by developers will be the same as those made by the subjects used in these studies and that the factors found to affect subjects performance will be the same as those that affect developers performance. This is a big assumption, because most studies of reasoning aim to uncover some aspect of *natural* human thinking and thus avoid using subjects who have had training in the formal (i.e., mathematical) use of logical. This contrasts with developers who will have had some training in the use of mathematical logic and a significant amount of reasoning experience through attempts to comprehend of code. The assumption stated above implies that the extent to which developers are more adept at deductive reasoning, compared to the general undergraduate population used in the studies (which may be different from the general population[1283]), is purely due to experience (formal training on its own does not usually affect performance; a study[224] of students who had taken a course in logic showed only a 3% improvement in performance, compared to those who had no formal logic training).

The three main topics discussed are:

memory 0
developer

1. use of working memory resources (for simplicity the use of long-term memory is not considered here),
2. the impact of the visual appearance of the source, and
3. the performance of humans when reasoning.

At the time of this writing the research results discussed raise questions, rather than providing answers. However, they do go a long way towards dispelling the notion that developer reasoning performance is independent of the logical form of a problem.

2 Remembering conditional information

How is the information denoted by a conditional expression represented in a readers mind? There are a number of possibilities, including:

- It is not represented at all. The source containing the controlling expression is reread on an as-needed basis.
- As some representation of the conditional expression appearing in the visible source. For instance, the spoken form of the expression (in which case the use of identifiers having shorter spoken forms would be an advantage).

```
1  if (character_count > 20)
2
3  if (char_count > 20) /* Contains fewer syllables, in spoken form, than above. */
```

- as what it represents internally, in the program, or what it represents within the application. For instance, the following two conditional expressions may both be represented in a readers mind using the same information.

```
1  if (character_count > MARGIN_WIDTH)
2
3  if (SKIPPED_PAST_MARGIN)
```

- some combination of the above. For instance, the conditional expression:

```
1  if ((ch >= MIN_PRINTABLE) && (ch <= MAX_PRINTABLE) &&
2      (ch != 'Q'))
```

might be represented, in a readers mind, as “a printable character that is not equal to the letter Q”.

3 Visual appearance

The bodies of **if** statements are commonly indented, visually, to the right of the **if** keyword. The greater the nesting of **if** statements the greater the indentation. This indenting practice has a cost impact that is separate from the cost of comprehending any relationship between the controlling expressions; the cost factors include:

- Reducing the visible line length reduces the probability that individual statements will fit on a single line. The issue of the readability of statements split over more than one line is discussed elsewhere. 1707 statement visual layout
- Each indentation creates a visually discriminable location that needs to be temporarily remembered by readers. Your author could not find any studies relating to this problem (the study by Hake[533] asked subjects to judge discriminate the position of a pointer on a scale, relative to two end points, using 5, 10, 20 and 50 different locations).

Situations requiring many that levels of nested **if** statements be written also often involve relatively large numbers of statements. Separating out the individual contributions made by indentation and number of lines (the issue of function size is discussed elsewhere) to the total comprehension effort, and being able to calculate the cost/benefit of the various possible source code organizations, requires a significantly greater amount of expertise than is currently available. For this reason these coding guidelines do not discuss these issues further. The issue of the visual layout of conditional expressions is discussed elsewhere. 1821 function definition syntax 940 expressions

4 Reasoning

It has been claimed that the ability to reason is what separates humans from the rest of the animal kingdom. However, studies[1209] of reasoning using illiterate subjects from remote parts of the world obtained answers to verbal reasoning problems that were based on personal experience and social norms, rather than the western ideal of logic. The answers given by subjects, in the same location, who had received several years of schooling were much more likely to match those demanded by mathematical logic; the subjects had learned to *play the game*. Peng, Ames, and Knowles[1071] discuss styles of reasoning used in various cultures.

Until relatively recently (Wason's famous four card selection task[1445] was first published in 1966; he did not at first question whether logic was the correct normative theory and interpreted the results in terms of people being illogical or irrational) it was believed that mathematical logic formed the basis for human rational thought (a belief in line with the model of the human mind as a general purpose computer).

An example of how a reader's performance can be affected by the kind of question asked, about conditions, is provided by a study by Bell and Johnson-Laird.[111] They asked subjects to give yes/no responses to two kinds of questions, asking about what is possible or what is necessary. They predicted that subjects would find it easier to infer a 'yes' answer to a question about what is possible, compared to one about what is necessary, because only one instance needs to be found, whereas all instances need to be checked to answer 'yes' to a question about necessity (see Table 1739.2). For instance, in a game in which only two can play:

```
If Allan is in then Betsy is in.
If Carla is in then David is out.
```

answering 'yes' to the question "Can Betsy be in the game?" (a possibility) is easier than giving the same answer to "Must Betsy be in the game?" (a necessity).

However, subjects would find it easier to infer a 'no' answer to a question about what is necessary, compared to one about what is possible, because only one instance needs to be found, whereas they all instances need to be checked to answer 'no' to a question about possibility. For instance, in another two person game:

```
If Allan is out then Betsy is out.
If Carla is out then David is in.
```

answering 'no' to the question "Must Betsy be in the game?" (a necessity) is easier than giving the same answer to "Can Betsy be in the game?" (a possibility).

Table 1739.2: Percentage of correct responses given to the four kinds of questions. Adapted from Bell and Johnson-Laird.[111]

Kind of Question	Correct 'yes' Response	Correct 'no' Response
is possible	91%	65%
is necessary	71%	81%

The study of reading and representing natural language sentences has to deal with the fact that such sentences in such languages are often syntactically and semantically ambiguous. Conditional statements written in the C language have a well defined syntactic and semantic meaning (we ignore the ill-formed cases here). However, the existence of a well defined meaning does not imply that all readers will find and use it.

5 Deductive reasoning

deductive reasoning

Heuristics and Biases

conjunction fallacy

pragmatic interpretation

A number of reasons for people's failure to give answers that matched those required by one of the mathematical logics (e.g., propositional or predicate calculus) have been proposed. These include the use of heuristics as a means of overcoming cognitive limits, interpreting the wording of questions in a pragmatic way based on how natural language are used rather than as logical formula, and interpreting the questions in a social context.[575] The following are some of the factors that have been found to affect peoples performance in solving deductive reasoning problems:

- *age of reasoner*— Performance in reasoning tasks declines with age.[427] Contributing factors for this decline include a reduction in working memory capacity with age[491] and a slowing down of cognitive processing speed.[1192]
- *Belief bias*— people have been found to be more willing to accept a conclusion, derived from given premises, that they believe to be true than one that they believe to be false. A study by Evans, Barston, and Pollard[402] gave subjects two premises and a conclusion and asked them to state whether the conclusion was true or false (based on the premises given). The conclusions were rated as either believable or unbelievable (this status was checked by asking a separate group of subjects to rate the believability of the conclusions on a seven point scale). The results showed (see Table 1739.3) that belief did affect performance, particularly when the conclusion was invalid.

Table 1739.3: Percentage of subjects accepting that the stated conclusion could be logically deduced from the given premises. Based on Evans, Barston, and Pollard.[402]

Status-context	Example	Conclusion Accepted
Valid-believable		
	No Police dogs are vicious Some highly trained dogs are vicious Therefore, some highly trained dogs are not police dogs	88%
Valid-unbelievable		
	No nutritional things are inexpensive Some vitamin tablets are inexpensive Therefore, some vitamin tablets are not nutritional things	56%
Invalid-believable		
	No addictive things are inexpensive Some cigarettes are inexpensive Therefore, some addictive things are not cigarettes	72%
Invalid-unbelievable		
	No millionaires are hard workers Some rich people are hard workers Therefore, some millionaires are not rich people	13%

- *Form of premise*— a study by Dickstein[354] measured subjects performance on the 64 possible two premise syllogisms (a premise being one of the propositions: *All S are P*, *No S are P*, *Some S are P*, and *Some S are not P*). For instance, the following syllogisms show the four possible permutations of three terms (the use of S and P is interchangeable):

```
All M are S      All S are M    All M are S     All S are M
No P are M       No P are M     No M are P      No M are P
```

  The results showed that performance was affected by the order in which the terms occurred (known as the *figure*) in the two premises of the syllogism. The order in which the premises are processed may affect the amount of working memory needed to reason about the syllogism, which in turn can affect human performance.[494]

- *Individual preferences*— different people have been found to prefer the use of different strategies when solving a deductive problem.

1739 reasoning strategy choice

Whatever the reason for people giving the answers they do, studies have shown that there are patterns to the *mistakes* made and that these patterns are found in a large number of people. A variety of theories and models have been proposed to explain these patterns of behavior. The following is a brief description of some of the different theories that have been proposed (the *mental model* theory currently enjoys general acclaim, based on its ability to predict the behavior seen in a large number of studies and the number of researchers currently publishing papers based on some form of it):

- *Mental logic.*[151, 1167] People perform logical reasoning by the application of formal (syntactic) rules. The larger the number of rules that are applied to infer a conclusion the more difficult the problem is.
- *Mental models.*[677, 678] Readers construct a mental model (or set of models) from the given premises (this is a two-stage process that involves comprehending a premise, followed by combining the information contained in each premise to form the model). This model is used to draw possible conclusions (i.e., people reason from the content of a problem, unlike mental logic where reasoning is based on the syntactic form) which are then subject to a process of validation. Validation involves searching for alternative models, or counter examples, that are consistent with the premises, but which refute the conclusion. A conclusion is considered to be valid if no counter example can be found. Errors in reasoning are the result of limitations in the processing ability of the mind, in particular:

  1. one model is better than many. That is, the fewer models needed for an inference, and the simpler they are, the easier the inference,
  2. reasoners sometimes fail to consider all models in multiple-model problems. This results in them drawing conclusions that are possible rather than necessary,
  3. reasoners focus on what is true and neglect what is false. This can result in illusionary inferences,
  4. premise contents and background knowledge can affect the interpretation of assertions made in a premise and the process of reasoning, and
  5. with experience, reasoners develop tailor-made strategies for particular sorts of problems. To refute invalid conclusions, they can search for counter-examples.

- *Darwinian algorithms.*[285] It is argued that being able to perform certain kinds of conditional reasoning offers an evolutionary advantage and that the human mind has innate rule structures for dealing with these kinds of problems. For instance, one of the basis for social exchange is the generic rule *If you take a benefit, you pay a cost.*
- *Information gain.*[1021] Here reasoners are said to have the goal of gaining information or reducing uncertainty (an adaption to the environment in which people traditionally have use reasoning). Conclusions are informative to the extent that they are improbable, or surprising.

- *Pragmatic reasoning schemas.*[223] These schemas are packages of knowledge about specific domains, containing rules for thought and action. Solving a particular problem requires matching production rules that evoke the appropriate schema.
- *Dual process theories.*[1257, 1283] People use two systems of reasoning (see Table 1739.4). Stanovich[1283] has shown a strong correlation between students performance on SAT tests (as well as intelligence tests) and the extent to which they are likely to use System 2 in reasoning tasks. The results of some studies[401] have suggested that people sometimes use both systems of reasoning when reasoning about a problem.

Table 1739.4: Properties of the two systems of thinking. Based on Stanovich.[1283]

System 1	System 2
Unconscious	Conscious
Automatic	Controlled
Associative	Rule-based
Heuristic processing	Analytic processing
Undemanding of cognitive capacity	Demanding of cognitive capacity
Relatively fast	Relatively slow
Acquisition by biology, exposure, and personal experience	Acquisition by cultural and formal training
Highly contextualized	Decontextualized
Conversational and socialized	Asocial
Independent of general intelligence	Correlated with general intelligence

Deductive reasoning problems can take many forms and this discussion limits itself to syllogisms (the simplest form and the one used by most studies). A syllogism consists of two premises and a conclusion. The two premises are usually given and the conclusion has to be deduced from them. The following are some of the different forms of syllogism:

- *Categorical syllogisms* use relations of inclusion. For instance, from

```
All A are B
All B are C
```

  we can deduce that "All A are C".

- *Linear syllogisms* use relations of order. For instance, from

```
A is taller than B
B is taller than C
```

  we can deduce that "A is taller than C".

- *Conditional syllogisms* use relations of implication. For instance, from

```
if A, then B
A is true
```

  we can deduce that "B is true".

These coding guidelines are aimed at the C programming language, which does not directly support operators for the operations that occur in categorical syllogisms (e.g., *all* or *some*). While developers may implement functions that perform equivalent operations these coding guidelines do not attempt to address developer defined operations. For this reason categorical syllogisms are not discussed further.

5.1 Linear reasoning

linear reasoning
1197 relational operators
syntax

The use of relational operators have an obvious interpretation in terms of linear syllogisms. A study by De Soto, London, and Handel[333] provides a good example. The task they used was based on what they called *social reasoning*, using the relations *better* and *worse*. Subjects were shown two premises, involving three people, and a possible conclusion (e.g., "Is Mantle worse than Moskowitz?"). They had 10 seconds to answer *yes*, *no*, or *don't know*. All four possible combinations of conclusions were used.

Table 1739.5: Eight sets of premises describing the same relative ordering between A, B, and C (peoples names were used in the study) in different ways, followed by the percentage of subjects giving the correct answer. Adapted from De Soto, London, and Handel.[333]

	Premises	Percentage Correct Response		Premises	Percentage Correct Response
1	A is better than B B is better than C	60.5	5	A is better than B C is worse than B	61.8
2	B is better than C A is better than B	52.8	6	C is worse than B A is better than B	57.0
3	B is worse than A C is worse than B	50.0	7	B is worse than A B is better than C	41.5
4	C is worse than B B is worse than A	42.5	8	B is better than C B is worse than A	38.3

Based on the results (see Table 1739.5) De Soto et al made two observations (which they called *paralogical principles*; cases 5 and 6 possess both, while cases 7 and 8 possess neither):

1. People learn orderings better in one direction than another. In this study people gave more correct answers when the direction was better-to-worse (case 1), than mixed direction (case 2, 3), and were least correct in the direction worse-to-better (case 4). This suggests that use of the word *better* should be preferred over *worse* (the British National Corpus[823] lists *better* as appearing 143 times per million words, while *worse* appears under 10 times per million words and is not listed in the top 124,000 most used words).
2. People end-anchor orderings. That is, they focus on the two extremes of the ordering. In this study people gave more correct answers when the premises stated an end term (better or worse) followed by the middle term, than a middle term followed by an end term.

A second experiment in the same study gave subjects printed statements about people. For instance, "Tom is better than Bill". The relations used were *better*, *worse*, *has lighter hair*, and *has darker hair*. The subjects had to write the peoples names in two of four possible boxes; two arranged horizontally and two arranged vertically.

The results showed 84% of subjects selecting a vertical direction for better/worse, with better at the top (which is consistent with the *up is good* usage found in English metaphors[797]). In the case of lighter/darker 66% of subjects used a horizontal direction, with no significant preference for left-to-right or right-to left.

A third experiment in the same study used the relations *to-the-left* and *to-the-right*, and *above* and *below*. The above/below results were very similar to those for better/worse. The left-right results showed that subjects learned a left-to-right ordering better than a right-to-left ordering.

The results of this study show the affect that operand order has on the accuracy of peoples responses. However, the interpretation placed on the operator also plays a significant role. It appears that without knowing what interpretation a reader is likely to give to the operands and operators in the following two (logically equivalent) conditional expressions, for instance, it is not possible to select the one that is most likely to minimize incorrect reasoning on the part of readers.

```
1  if ((x <= y) && (x => z))
2  if ((x >= z) && (x <= y))
```

Since the De Soto, et al. study additional factors have been discovered and a number of models have been proposed to explain the strategies used by people in solving linear reasoning problems. These include:

- *The spatial model*[333, 609]— people integrate information from each premise into a spatial array representing all known relationships.
- *The linguistic model*[241]— people represent each premise using linguistic propositions (the individual premises are not integrated).
- *The algorithmic model*[1133]— people apply some algorithm to the structure of the linguistic representation of the premises. For instance, given "Reg is taller than Jason; Keith is shorter than Jason" and the question "Who is the shortest?", a so called *elimination strategy* was used by some subjects in the study. (The answer for the first premise is Jason, which eliminates Reg; the answer to the second premise is Keith which eliminates Jason, so Keith is the answer).
- *The mixed model*[1292]— the information in the premise is first decoded into a linguistic form and then encoded into a spatial form.

reasoning strategy choice

The strategy used to solve a given problem has been found to vary between people. A study by Sternberg and Weil[1293] found a significant interaction between a subjects' aptitude (as measured by verbal and spatial ability tests) and the strategy they used to solve linear reasoning problems. However, a person having high spatial ability, for instance, does not necessarily use a spatial strategy. A study by Roberts, Gilmore, and Wood[1173] asked subjects to solve what appeared to be a spatial problem (requiring the use of a very inefficient spatial strategy to solve). Subjects with high spatial ability used non-spatial strategies, while those with low spatial ability used a spatial strategy. The conclusion made was that those with high spatial ability were able to see that the spatial strategy was inefficient to select as alternative strategy, while those with less spatial ability were unable to perform this evaluation.

A study by Mayer[905] asked subjects to learn a sequence of facts expressed as a relationship, such as $B > C$, $A > B$, and $A > C$ (the ordering $A > B > C > D > E > F$ was true for all facts the subjects were asked to learn). One group of subjects were told to think of the operands as boys names and the relation as representing *taller than*, while the other group were given no such instruction. Once the subjects had learned the facts presented to them, they were tested by having to answer questions about them. The results suggested that subjects who had been told to think about the relationship as representing *taller than* had integrated the separately presented facts into a single linear-encoding. While the other subjects did not integrate the separate facts and unencoded each fact using a single association.

The information encoding used by people can affect how well they later recall that information and also their performance when making comparisons about objects or symbols having some attribute that varies along a continuem. For instance, studies[1114] have found that subjects performance improves the further apart the two objects being compared are perceived to be.

symbolic distance effect 1197

5.2 Causal reasoning

A question often asked by developers, while reading source, is "what causes this event/situation to occur?" Causal questions such as this are also common in everyday life. However, there has been little mathematical research (statistics deals with correlation) on causality (Pearl[1067] is an exception) and little psychological research on causal reasoning. It is often possible to express a problem in either a causal or conditional form. A study by Sloman, and Lagnado[1256] gave subjects one of the following two reasoning problems and associated questions:

- *Causal argument* form:

```
A causes B
A causes C
B causes D
C causes D
D definitely occurred
```

with the questions: "If B had not occurred, would D still have occurred?", or "If B had not occurred, would A have occurred?".

- *Conditional argument* form:

```
If A then B
If A then C
If B then D
If C then D
D is true
```

with the questions: "If B were false, would D still be true?", or "If B were false, would A be true?".

The results (see Table 1739.6) showed that subject performance depended on the form in which the problem was expression.

Table 1739.6: Percentage *yes* responses to various forms of questions (based on 238 responses). Based on Sloman, and Lagnado.[1256]

Question	Causal	Conditional
D holds?	80%	57%
A holds?	79%	36%

There has been relatively little psychological research into causality and counter factual reasoning, although this is starting to change. This subsection is intended to show that human performance with this kind of reasoning may not be the same as that when using conditional reasoning.

5.3 Conditionals in English

In all natural languages, the conditional clause generally precedes the conclusion, in a conditional statement.[264] An example where the conditional follows the conclusion is "I will leave, if you pay me" given as the answer to the question "Under what circumstances will you leave?". In one study of English[435] the conditional preceded the conclusion in 77% of written material and 82% of spoken material.

Table 1739.7: Occurrence of the most common conditional sentence types in speech (266 conditionals from a 63,746 word corpus) and writing (948 conditionals from 357,249 word corpus). In the notation *if* + *x*, *y*: *x* is the condition (which might, for instance, be in the past tense) and *y* can be thought of as the *then part* (which might, for instance, use one of the words would/could/might, or be in the present tense). Adapted from Celce-Murcia.[213]

Structure	Speech	Writing
If + present, present	19.2	16.5
If + present, (will/be going to)	10.9	12.5
If + past, (would/might/could)	10.2	10.0
If + present, (should/must/can/may)	9.0	12.1
If + (were/were to), (would/could/might)	8.6	6.0
If + (had/have +en), (would/could/might) have	3.8	3.3
If + present, (would/could/might)	2.6	6.1

Table 1739.8: Occurrence of various kinds of **if** statement controlling expressions (as a percentage of all **if** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *integer-constant* is an integer constant expression, and *expression* represents all other expressions. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	32.4	! function-call	3.8
object	15.5	object < *integer-constant*	2.2
object == object	8.9	object > *integer-constant*	1.8
! object	7.4	function-call == object	1.6
function-call	7.4	object > object	1.4
expression	5.7	object != *integer-constant*	1.3
object != object	4.2	function-call == *integer-constant*	1.2
object == *integer-constant*	4.0	object < object	1.1

Table 1739.9: Occurrence of various kinds of **switch** statement controlling expressions (as a percentage of all **switch** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
object	75.3
function-call	14.2
expression	5.2
others	3.3
*v object	2.0

Table 1739.10: Occurrence of equality, relational, and logical operators in the conditional expression of an **if** statement (as a percentage of all such controlling expressions and as a percentage of all occurrences each operator in the source). Based on the visible form of the `.c` files. The percentage of controlling expressions may not sum to 100% because more than one of the operators occurs in the same expression.

Operator	% Controlling Expression	% Occurrence of Operator	Operator	% Controlling Expression	% Occurrence of Operator
==	31.7	88.6	>=	3.5	76.8
!=	14.1	79.7	no relational/equality	47.5	—
<	6.9	45.6	||	9.6	85.9
<=	1.9	68.6	&&	14.5	82.3
>	3.5	84.9	no logical operators	84.2	—

Semantics

controlling expression
if statement

A selection statement selects among a set of statements depending on the value of a controlling expression. 1740

Commentary

terms 31 defined where

The term *controlling expression* does not appear in italic type in the Standard. This C sentence might be considered to be its first definition.

C++

The C++ Standard omits to specify how the flows of control are selected:

6.4p1

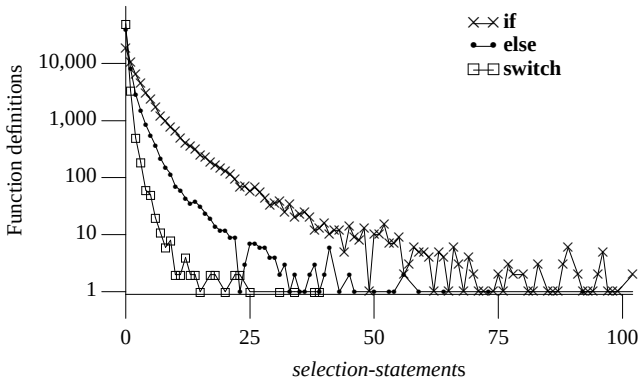


Figure 1739.1: Number of function definitions containing a given number of `selection-statement`s. Based on the translated form of this book's benchmark programs.

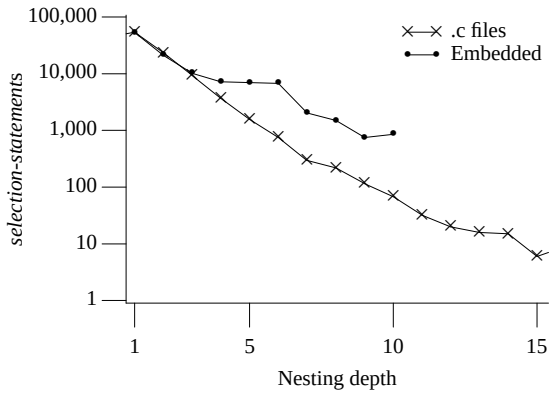


Figure 1739.2: Number of `selection-statement`s having a given maximum nesting level for embedded C[389] (whose data was multiplied by a constant to allow comparison; the data for nesting depth 5 was interpolated from the adjacent points). Based on the visible form of the `.c` files.

Selection statements choose one of several flows of control.

Coding Guidelines

The evaluation of a controlling expression is used to select the flow of control. The issue of side effects occurring during the evaluation is frequently discussed by developers and writers of coding guideline documents. Experience suggests that the following are the primary reasons for developers to write controlling expressions containing side effects:

- A belief that the resulting machine code is more efficient. In many cases this belief is false. For instance, most translators generate the same machine code generated for:

```
1   if (x = y)
```

  and:

```
1   x=y;
2   if (x != 0)
```

- Reducing the effort needed to organize the layout of the visible source, when writing it. For instance, an assignment inside a series of nested **`if`** statements would require the use of braces:

```
if (x = y)
   /* ... */
```

might have to be written as:

```
{
x=y;
if (x != 0)
   /* ... */
}
```

Neither of these reasons could be said to contain an actual benefit. The cost associated with side effects in controlling expressions is the possibility that they will go unnoticed by a reader of the source (especially if scanning along the left edge looking for assignments).

reading 770 kinds of

The most common form of side effect in a controlling expression is assignment, in particular simple assignment. The case where the author of the code intended to type an equality operator, rather than a simple assignment operator is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. However, it is possible that a reader of the visible source will mistake a simple assignment for an equality operator (the token **==** is much more likely than **=** in the context of a controlling expression) and reducing the likelihood of such a mistake occurring is also a cost reduction.

guidelines 0 not faults

controlling 1740 expression if statement

This discussion has referred to controlling expressions as if these costs and benefits apply to their use in all contexts (i.e., selection and iteration statements). The following example shows that writing code to avoid the occurrence of side effects in controlling expressions contained with iteration statements requires two, rather than one, assignments to be used.

```
extern int glob_1,
           glob_2;

void f_1(void)
{
if (glob_1 = glob_2)
   ;
while ((glob_1 = glob_2 + 1) != 3)
   { /* ... */ }
}

void f_2(void)
{
   {
   glob_1 = glob_2;     /* Single statement. */
   if (glob_1 != 0)
      ;
   }

glob_1 = glob_2 + 1;    /* Statement 1: always occurs. */
while (glob_1 != 3)
   {
   /* ... */
   glob_1 = glob_2 + 1; /* Statement 2: occurs after every iteration. */
   }
}
```

Duplicating the assignment to `glob_1` creates a maintenance dependency (any changes to one statement need to be reflected in the other). The increase in cost caused by this maintenance dependency is assumed to be greater than the cost reduction achieved from reducing the likelihood of a simple assignment operator being mistaken treated as an equality operator.

Cg 1740.1 The simple assignment operator shall not occur in the controlling expression of an **if** statement.

Experience has shown that there are a variety of other constructs, appearing in a controlling expression, that developer have difficulty comprehending, or simply miscomprehend when scanning the source. However, no other constructs are discussed here. The guideline recommendation dealing with the use of the assignment operator has the benefit of simplicity and frequency of occurrence. It was difficult enough analyzing the cost/benefit case for simple assignment and others are welcome to address more complicated cases.

Experience shows that many developers use the verbal form "if expression is not true then" when thinking about the condition under which an **else** form is executed. This use of *not* can lead to double negatives when reading some expressions. For instance, possible verbal forms of expressing the conditions under which the arms of an **if** statement are executed include:

```
1  if (!x)
2     a(); /* Executed if not x.               */
3  else
4     b(); /* Executed if not x is not true.   */
5          /* Executed if not x is equal to 0. */
6          /* Executed if x is not equal to 0. */
7
8  if (x != y)
9     c(); /* Executed if x is not equal to y.             */
10 else
11    d(); /* Executed if x is not equal to y is not true. */
```

The possible on linguistic impact of the ! operator on expression comprehension is discussed elsewhere. 1103 ! operand type

Cg 1740.2

The top-level operator in the controlling expression of an **if** statement shall not be ! or != when that statement also contains an **else** arm.

If the value of the controlling expression is known a translation time, the selection statement may contain dead code and the controlling expression is redundant. These issues are discussed elsewhere. 190 dead code 190 redundant code

Usage

In the translated form of this book's benchmark programs 1.3% of *selection-statement*s and 4% of *iteration-statement*s have a controlling expression that is a constant expression. Use of simple, non-iterative, flow analysis enables a further 0.6% of all controlling expressions to be evaluated to a constant expression at translation time.

1741 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. block selection statement

Commentary

```
enum {a, b};

int different(void)
{
    if (sizeof(enum {b, a}) != sizeof(int))
        return a;  // a == 1
    return b;  // which b?
}
```

Rationale

In C89, the declaration enum {b, a} persists after the if-statement terminates; but in C99, the implied block that encloses the entire **if** statement limits the scope of that declaration; therefore the different function returns different values in C89 and C99. The Committee views such cases as unintended artifacts of allowing declarations as operands of cast and **sizeof** operators; and this change is not viewed as a serious problem.

See the following C sentence for a further discussion on the rationale. 1742 block selection sub-statement

C90

See Commentary.

C++

The C++ behavior is the same as C90. See Commentary.

Coding Guidelines

Developers are more likely to be tripped up by the lifetime issues associated with compound literals than enumeration constants. For instance in:

```
1  if (f(p=&(struct S){1, 2}))
2     /* ... */
3  val=p->mem_1;
```

the lifetime of the storage whose address is assigned to `p` ends when the execution of the **`if`** statement terminates. Ensuring that developers are aware of this behavior is an educational issue. However, developers intentionally relying on the pointed-to storage continuing to exist (which it is likely to, at least until storage needs to be allocated to another object) is a potential guideline issue. However, until experience has been gained on how developers use compound literals it is not known whether this issue is simply an interesting theoretical idea of a real practical problem.

---

block selection sub-statement

Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement. 1742

Commentary

Rationale

A new feature of C99: A common coding practice is always to use compound statements for every selection and iteration statement because this guards against inadvertent problems when changes are made in the future. Because this can lead to surprising behavior in connection with certain uses of compound literals (§6.5.2.5), the concept of a block has been expanded in C99.

Given the following example involving three different compound literals:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    if (*(q = (int[2]){i, j}))
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    else
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);

    return *p;
}
```

it seemed surprising that just introducing compound statements also introduced undefined behavior:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;
```

```
    if (*(q = (int[2]){i, j})) {
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    } else {
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
    }

    return *p;  // undefined--no guarantee *p designates an object
}
```

Therefore, the substatements associated with all selection and iteration statements are now defined to be blocks, even if they are not also compound statements. A compound statement remains a block, but is no longer the only kind of block. Furthermore, all selection and iteration statements themselves are also blocks, implying no guarantee that `*q` in the previous example designates an object, since the above example behaves as if written:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    {
        if (*(q = (int[2]){i, j})) {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){9, 8, 7, 6, 5}, q);
        } else {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
        }
    }

    // *q is not guaranteed to designate an object

    return *p;  // *p is not guaranteed to designate an object
}
```

If compound literals are defined in selection or iteration statements, their lifetimes are limited to the implied enclosing block; therefore the definition of “block” has been moved to this section. This change is compatible with similar C++ rules.

C90

The following example illustrates the rather unusual combination of circumstances needed for the specification change, introduced in C99, to result in a change of behavior.

```
1  extern void f(int);
2  enum {a, b} glob;
3
4  int different(void)
5  {
6  if (glob == a)
7                        /* No braces.                                          */
8     f((enum {b, a})1); /* Declare identifiers with same name and compatible type. */
9
10 return b;  /* C90: numeric value 1 */
11            /* C99: numeric value 0 */
12 }
```

C++

The C++ behavior is the same as C90.

Coding Guidelines

using braces
block
compound 1729
statement
syntax

Some coding guideline documents recommend that the block associated with both selection and iteration statements always be bracketed with braces (i.e., that it is always a compound statement). When the compound statement contains a single statement the use of braces is redundant and their presence decreases the amount of information visible on a display (the number of available lines is fixed and each brace usually occupies one line). However, experience has shown that in some cases the presence of these braces can:

- Provide additional visual cues that can reduce the effort needed, by readers, to comprehend a sequence of statements. However, the presence of these redundant braces reduces the total amount of information immediately visible, to a reader, on a single display (i.e., the amount of source code that can be seen without expending motor effort giving editor commands to change the display contents). The way in which these costs and benefits trade-off against each other is not known.

cost/accuracy 0
trade-off

- Help prevent faults being introduced when code is modified (i.e., where a modification results in unintended changes to the syntactic bindings of blocks to statement headers). Experience shows that nested **if** statements are the most common construct whose modification results in unintended changes to the syntactic bindings of blocks.

statement 1707
header

In the following example the presence of braces provides both visual cues that the **else** does not bind to the outer **if** and additional evidence (its indentation provides counter evidence because it provides an interpretation that the intent is to bind to the inner **if**.

```
1  void f(int i)
2  {
3  if (i > 8)
4     if (i < 20)
5        i++;
6  else
7     i--;
8
9  if (i > 8)
10    {
11    if (i < 20)
12       i++;
13 else
14    i--;
15    }
16 }
```

Blocks occur in a number of statements, is there a worthwhile cost/benefit in guideline recommendation specifying that these blocks always be a compound statement?

The block associated with a **switch** statement is invariably a compound statement. A guideline recommendation that braces be used is very likely to be redundant in this case. Iteration statements are not as common as selection statements and much less likely to be nested (in other iteration statements) than selection statements (compare Figure 1739.2 and Figure 1763.2), and experience suggests developer comprehension of such constructs is significantly affected by the use of braces. Experience suggests that the nested **if** statement is the only construct where the benefit of the use of braces is usually greater than the cost.

Cg 1742.1

The statement forming the block associated with either arm of an **if** statement shall not be an **if** statement.

6.8.4.1 The if statement

Constraints

1743 The controlling expression of an **if** statement shall have scalar type.

if statement controlling expression scalar type

Commentary

Although the type **_Bool** was introduced in C99 the Committee decided that there was too much existing code to change the specification for the controlling expression type.

C++

The value of a condition that is an expression is the value of the expression, implicitly converted to ***bool*** *for statements other than* ***switch****; if that conversion is ill-formed, the program is ill-formed.* 6.4p4

If only constructs that are available in C are used the set of possible expressions is the same.

Other Languages

In many languages the controlling expression is required to have a boolean type. Languages whose design has been influenced by C often allow the controlling expression to have scalar type.

Coding Guidelines

The broader contexts in which readers need to comprehend controlling expression are discussed elsewhere. This subsection concentrates on the form of the controlling expression. 1739 selection statement syntax

The value of a controlling expression is used to make one of two choices. Values used in this way are generally considered to have a boolean role. Some languages require the controlling expression to have a boolean type and their translators enforce this requirement. Some coding guideline documents contain recommendations that effectively try to duplicate this boolean type requirement found in other languages. Recommendations based on type not only faces technical problems in their wording and implementation (caused by the implicit promotions and conversions performed in C), but also fail to address the real issues of developer comprehension and performance.

In the context of an **if** statement do readers of the source distinguish between expressions that have two possible values (i.e., boolean roles), and expressions that may have more than two values being used in a context where an implicit test against zero is performed? Is the consideration of boolean roles a cultural baggage carried over to C by developers who have previously used them in other languages? Do readers who have only ever programmed in C make use of boolean roles, or do they think in terms of *a test against zero*? In the absence of studies of developer mental representations of algorithmic and source code constructs, it is not possible to reliably answer these questions. Instead the following discussion looks at the main issues involved in making use of boolean roles and making use of the implicit *a test against zero* special case.

A boolean role is not about the type of an expression (prior to the introduction of the type **_Bool** in C99, a character type was often used as a stand-in), but about the possible values an expression may have and how they are used. The following discussion applies whether a controlling expression has an integer, floating, or pointer type.

In some cases the top-level operator of a controlling expression returns a result that is either zero or one (e.g., the relational and equality operators). The visibility, in the source, of such an operator signals its boolean role to readers. However, in other cases (see Table 1763.2) developers write controlling expressions that do not contain explicit comparisons (the value of a controlling expression is implicitly compared against zero). What are the costs and benefits of omitting an explicit comparison? The following code fragment contains examples of various ways of writing a controlling expression:

```
1  if (flag)            /* 1 */
2     /* ... */
3
4  if (int_value)       /* 2 */
5     /* ... */
6
7  if (flag == TRUE)    /* 3 */
8     /* ... */
9
10 if (int_value != 0)  /* 4 */
11    /* ... */
```

Does the presence of an explicit visual (rather than an implicit, in the developers mind) comparison reduce either the cognitive effort needed to comprehend the **if** statement or the likelihood of readers making mistakes? Given sufficient practice readers can learn to automatically process `if (x)` as if it had been written as `if (x != 0)`. The amount of practice needed to attain an automatic level of performance is unknown. Another unknown is the extent to which the token sequence `!= 0` acts as a visual memory aid.

When the visible form of the controlling expression is denoted by a single object (which may be an ordinary identifier, or the member of a structure, or some other construct where a value is obtained from an object) that name may provide information on the values it represents. To obtain this information readers might make use of the following:

- *Software development conventions.* In the software development community (and other communities) the term *flag* is generally understood to refer to something that can be in one of two states. For instance, the identifier `mimbo_flag` is likely to be interpreted as having two possible values relating to a *mimbo*, rather than referring to the national flag of Mimbo. Some naming conventions contain a greater degree of uncertainty than others. For instance, identifiers whose names contain the character sequence *status* sometimes represent more than two values.
- *Natural language knowledge.* Speakers of English regard some prepositions as being capable of representing two states. For instance, a cat *is* or *is not* black. This natural language usage is often adopted when selecting identifier names. For instance, `is_flobber` is likely to be interpreted as representing one of two states (being a, or having the attribute of, *flobber* or not).
- *Real world knowledge.* A program sometimes needs to take account of information from the *real world*. For instance, height above the ground is an important piece of information in an airplane flight simulator, with zero height having a special status.
- *Application knowledge.* The design of a program invariably makes use of knowledge about the application domain it operates within. For instance, the term `ziffer` may be used within the application domain that a program is intended to operate. Readers of the source will need the appropriate application knowledge to interpret the role of this identifier.
- *Program implementation conventions.* The design of a program involves creating and using various conventions. For instance, a program dealing with book printing may perform special processing for books that don't contain any pages (e.g., `num_pages` being zero is a special case).
- *Conventions and knowledge from different may be mixed together.* For instance, the identifier name `current_color` suggests that it represents color information. This kind of information is not usually thought about in terms of numeric values and there are certainly more than two colors. However, assigning values to symbolic qualities is a common software development convention, as is assigning a special interpretation to certain values (e.g., using zero to represent no known color, a program implementation convention).

The likelihood of a reader assuming that an identifier name has a boolean role will depend on the cultural beliefs and conventions they share with the author of the source. There is also the possibility that rather than

using the identifier name to deduce a boolean role, readers may use the context in which it occurs to infer a boolean role. This is an example of trust based usage. Requiring that values always be compared (against true/false or zero/nonzero) leads to infinite regression, as in the sequence:

476 boolean role
792 trust based usage

```
1  if (flag)
2  if (flag == TRUE)
3  if ((flag == TRUE) == TRUE)
4  and so on...
```

At some point readers have to make a *final* comparison in their own mind. The inability to calculate (i.e., automatically enforceable) the form a controlling expression should take to minimize readers cognitive effort prevents any guideline recommendations being made here.

Semantics

1744 In both forms, the first substatement is executed if the expression compares unequal to 0.

if statement operand compare against 0

Commentary

Depending on the type of the other operand this 0 may be converted to an integer type of greater rank, a floating-point `0.0`, or a null pointer constant.

748 null pointer constant

C++

The C++ Standard expresses this behavior in terms of true and false (6.4.1p1). The effect is the same.

Other Languages

In languages that support a boolean type this test is usually expressed in terms of **`true`** and **`false`**.

Common Implementations

The machine code generation issues are similar to those that apply to the logical operators. The degree to which this comparison can be optimized away depends on the form of the controlling expression and the processor instruction set. If the controlling expressions top-level operator is one that always returns a value of zero or one (e.g., an equality or relational operator), it is possible to generate machine code that performs a branch rather than returning a value that is then compared. Some processors have a single instruction that performs a comparison and branch, while others have separate instructions (the comparison instruction setting processor condition flags that are then tested by a conditional branch instruction). On some processors simply loading a value into a register also results in a comparison against zero being made, with the appropriate processor condition flags being set. The use of conditional instructions is discussed elsewhere.

1111 logical negation result is
1250 && operand compare against 0
1739 conditional instructions

The machine code for the first substatement is often placed immediately after the code to evaluate the controlling expression. However, optimizers may reorder blocks of code in an attempt to maximize instruction cache utilization.

1710 basic block

1745 In the **`else`** form, the second substatement is executed if the expression compares equal to 0.

else

Commentary

Implementations are required to ensure that exactly one of the equality comparisons is true.

1221 equality operators exactly one relation is true

Coding Guidelines

Some coding guideline documents recommend that the **`else`** form always be present, even if it contains no executable statements. Such a recommendation has the benefit of ensuring that there are never any mismatching **`if`**/**`else`** pairs. However, then the same effect can be achieved by requiring nested **`if`** statements to be enclosed in braces (this issue is discussed elsewhere). The cost of adding empty **`else`** forms increases the amount of source code that may need to be read and in some cases decrease in the amount of non-null source that appears on a display device. Such a guideline recommendation does not appear worthwhile.

1742 using braces block

Usage

In the visible form of the .c files 21.5% of **if** statements have an **else** form. (Counting all forms of *if* supported by the preprocessor, with **#elif** counting as both an *if* and an *else*, there is an **#else** form in 25.0% of cases.)

If the first substatement is reached via a label, the second substatement is not executed. 1746

Commentary

The flow of control of a sequence of statements is not influenced by how they were initially reached, in the flow of control. The label may be reached as a result of executing a **switch** statement, or a **goto** statement. The issue of jumping into nested blocks or the body of iteration statements is discussed elsewhere.

switch 1753 statement causes jump
goto 1789 causes unconditional jump
jump 1783 statement causes jump to
iteration 1766 statement executed repeatedly

C++

The C++ Standard does not explicitly specify the behavior for this case.

Other Languages

This statement applies to all programming languages that support jumps into more deeply nested blocks.

---

else binds to nearest if

An **else** is associated with the lexically nearest preceding **if** that is allowed by the syntax. 1747

Commentary

selection 1739 statement syntax

As it appears in the standard the syntax for **if** statements is ambiguous on how an **else** should be associated in a nested **if** statement. This semantic rule resolves this ambiguity.

Other Languages

Languages that support nesting of conditional statements need a method of resolving which construct an **else** binds to. The rules used include the following:

- Not supporting in the language syntax unbracketed nesting (i.e., requiring braces or **begin**/**end** pairs) within the *then* arm. For instance, Algol 60 permits the usage `IF q1 THEN a1 ELSE IF q2 THEN a2 ELSE a3`, but the following is a syntax violation `IF q1 THEN IF q2 THEN a1 ELSE a2 ELSE a3`.
- Using a matching token to pair with the **if**. The keyword **fi** is a common choice (used by Ada, Algol 68, while the C preprocessor uses **endif**). In this case the bracketing formed by the **if**/**fi** prevents any ambiguity occurring.
- Like C— using the *nearest preceding* rule.

Coding Guidelines

if statement 1742.1 block not an if statement

If the guideline recommendation on using braces is followed there will only ever be one lexically preceding **if** that an **else** can be associated with. Some coding guideline documents recommend that an **if** statement always have an associated **else** form, even if it only contains the null statement.

null statement 1733

6.8.4.2 The switch statement

Constraints

---

switch statement

The controlling expression of a **switch** statement shall have integer type. 1748

Commentary

A **switch** statement uses the exact value of its controlling expression and it is not possible to guarantee the exact value of an expression having a floating type (there is a degree of unpredictability in the value between different implementations). For this reason implementations are not required to support controlling expressions having a floating type.

C++

> *The condition shall be of integral type, enumeration type, or of a class type for which a single conversion function to integral or enumeration type exists (12.3).* 6.4.2p2

If only constructs that are available in C are used the set of possible expressions is the same.

Common Implementations

The base document did not support the types **long** and **unsigned long**. Support for integer types with rank greater than **int** was added during the early evolution of C.[1180]

Other Languages

There are some relatively modern languages (e.g., Perl) that do not support a **switch** statement. Java does not support controlling expressions having type **long**. Some languages (e.g., PHP) support controlling expressions having a string type.

Coding Guidelines

A controlling expression, in a **switch** statement, having a boolean role might be thought to be unusual, an **if** statement being considered more appropriate. However, the designer may be expecting the type of the controlling expression to evolve to a non-boolean role, or the **switch** statement may have once contained more **case** labels.

Table 1748.1: Occurrence of **switch** statements having a controlling expression of the given type (as a percentage of all **switch** statements). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	29.5	bit-field	3.1
unsigned long	18.7	**unsigned short**	2.8
enum	14.6	**short**	2.5
unsigned char	12.4	**long**	0.9
unsigned int	10.0	other-types	0.2
char	5.1		

1749 If a **switch** statement has an associated **case** or **default** label within the scope of an identifier with a variably modified type

switch past variably modified type

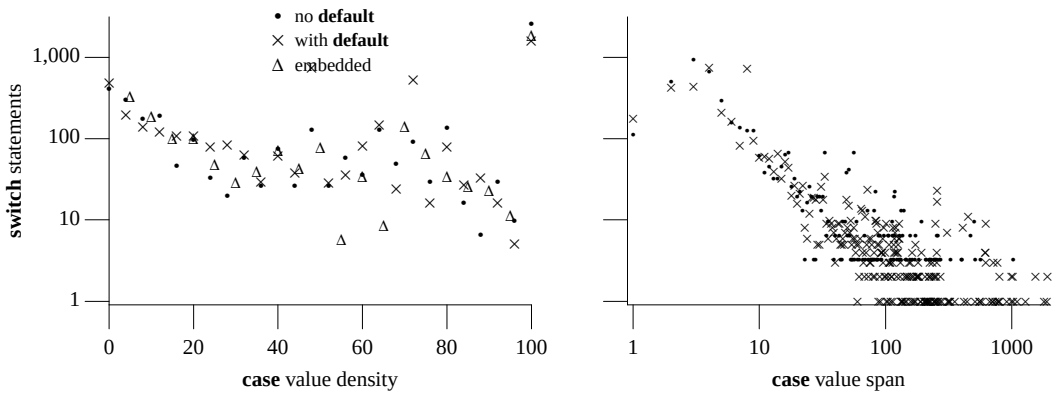


Figure 1748.1: *Density* of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one) divided by the number of **case** labels associated with a **switch** statement) and span of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one)). Based on the translated form of this book's benchmark programs and embedded results from Engblom[389] (which were scaled, i.e., multiplied by a constant, to allow comparison). The *no* ***default*** results were scaled so that the total count of **switch** statements matched those that included a **default** label.

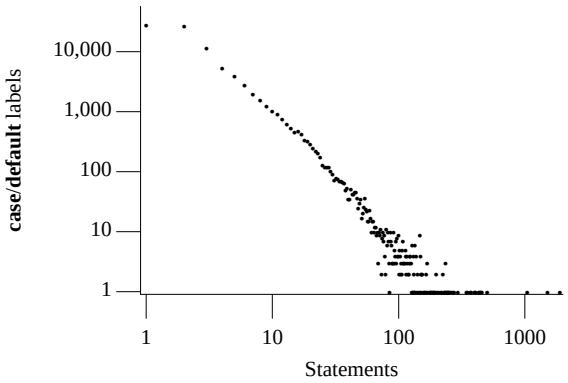


Figure 1748.2: Number of **case/default** labels having s given number of statements following them (statements from any nested **switch** statements did not contribute towards the count of a label). Based on the visible form of the `.c` files.

modified type, the entire **switch** statement shall be within the scope of that identifier.[133)]

Commentary

The declaration of an identifier having variable modified type can occur in one of the sequence of statements labeled by a **case** or **default**, provided it appears within a compound statement that does not contain any other **case** or **default** labels associated with that **switch** statement, or it appear after the last **case** or **default** label in the **switch** statement. In the compound statement case the variably modified type will not be within the scope of any **case** or **default** labels (its lifetime terminates at the end of the compound statement).

The wording of the requirement is overly strict in that it prohibits uses that might be considered well behaved. For instance:

```
1   switch (i)
2      {
3      case 1:
4            int x[n];
5            /* ... */
6            break;
7
8      case 2:
9            /* Statements that don't access x. */
10     }
```

Attempting to create wording to support such edge cases was considered to be a risk (various ambiguities may later be found in it) that was not worth the benefit. Additional rationale for this requirement is discussed elsewhere.

goto 1788 past variably modified type

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

The C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that being visible rather than *in scope* is more accurate terminology):

6.7p3 *It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps[77)] from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).*

```
void f(void)
{
switch (2)
   {
   int loc = 99; /* strictly conforming */
                 // ill-formed

   case 2: return;
   }
}
```

Example

```
extern int glob;

void f(int p_loc)
{
switch (p_loc) /* This part of the switch statement is not within the scope of a_1. */
   {
   case 1: ;
           int a_1[glob]; /* This declaration causes a constraint violation. */

   case 2: a_1[2] = 4;
           break;

   case 3: {
           long a_2[glob]; /* Conforming: no case label within the scope of a_2. */
           /* ... */
           }
           break;
   }
}
```

1750 The expression of each **case** label shall be an integer constant expression and no two of the **case** constant expressions in the same **switch** statement shall have the same value after conversion.

case label unique in same switch

Commentary

Two case labels having the same value is effectively equivalent to declaring two labels, within the same function, having the same name.

Coding Guidelines

Some sequences of case label values might be considered to contain suspicious entries or omissions. For instance, a single value that is significantly larger or smaller than the other values (an island), or a value missing from the middle of a contiguous sequence of values (a hole). While some static analysis tools check for such *suspicious values*, it is not clear to your author what, if any, guideline recommendation would be worthwhile.

1751 There may be at most one **default** label in a **switch** statement.

default label at most one

Commentary

A **default** label is the destination of a jump for some, possible empty, set of values of the controlling expression. As such it is required to be unique (if it occurs) within a **switch** statement.

A bug in the terminology being used in the standard "may" ⇒ "shall".

Coding Guidelines

Some coding guideline documents recommend that all **switch** statements contain a **default** label. There does not appear to be an obvious benefit (as defined by these coding guideline subsections, although there may be benefits for other reasons) for such a guideline recommendation. To adhere to the guideline developers simply need to supply a **default** label and an associated null statement. There are a number of situations where adhering to such a guideline recommendation leads to the creation of redundant code (e.g., if all possible values are covered by the **case** labels, either because they handle all values that the controlling expression can take or because execution of the **switch** statement is conditional on an **if** statement that guarantees the controlling expression is within a known range).

redundant code 190

Usage

In the visible form of the `.c` files, 72.8% of **switch** statements contain a **default** label.

(Any enclosed **switch** statement may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.) 1752

Commentary

This specification (semantics in a Constraints clause) clarifies the interpretation to be given to the phrase "in the same **switch** statement" appearing earlier in this Constraints clause.

case label unique in same switch 1750

Semantics

---

switch statement causes jump

A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. 1753

Commentary

This defines the term *switch body*. Developers also use the terminology *body of the switch*.

It is possible to write a **switch** statement as an equivalent sequence of **if** statements. However, experience shows that in some cases the **switch** statement appears to require less significantly less (cognitive) effort to comprehend than a sequence of **if** statements.

Common Implementations

Many processors include some form of instruction (often called an *indirect jump*) that indexes into a table (commonly known as a *jump table*) to obtain a location to jump to. The extent to which it is considered to be more efficient to use such an instruction, rather than a series of **if** statements varies between processors (whose behavior varies for the case where the index is out of range of the jump table) and implementations (the sophistication of the available optimizer). The presence of a **default** label creates additional complications in that all values of the controlling expression, not covered by a **case** label, need to be explicitly handled. Spuler[1275] discusses the general issues.

Some translators implement **switch** statements as a series of **if** statements. Knowledgeable developers know that, in such implementations, placing the most frequently executed **case** labels before the less frequently executed ones can provide a worthwhile performance improvement. Some translators[20,578] provide an option that allows the developer to specify whether a jump table or sequence of **if** statements should to be used.

Optimal execution time performance is not the only factor that implementations need to consider. The storage occupied by the jump table sometimes needs to be taken into account. In a simple implementation it is proportional to the difference between the maximum and minimum values appearing in the **case** labels (which may not be considered an efficient use of storage if there are only a few case labels used within this range). A more sophisticated technique than using a series of **if** statements is to create a binary tree of **case** label values and jump addresses. The value of the controlling expression being used to walk this tree to obtain the destination address. Some optimizers split the implementation into a jump table for those **case** label values that are contiguous and a binary tree for the out lying values.

Translator vendors targeting modern processors face an additional problem. Successful processors often contain a range of different implementations, creating a processor family, (e.g., the Intel Pentium series). These different processor implementations usually have different performance characteristics, and in the case of the **switch** statement different levels of sophistication in branch prediction. How does a translator make the decision on whether to use a jump table or **if** statements when the optimal code varies between different implementations of a particular processor?

A study by Uh and Whalley[1387] compared (see Table 1753.1) the performance of a series of **if** statements and the equivalent jump table implementation. For three of the processors it was worth using a jump table when there were more than two **if** statements were likely to be executed. In the case of the UltraSPARC-1 the figure was more than eight **if** statements executed (this was put down to the lack hardware support for branch prediction of indirect jumps).

Table 1753.1: Performance comparison (in seconds) of some implementation techniques for a series of **if** statements (contained in a loop that iterated 10,000,000 times) using (1) linear search (LS), or (2) indirect jump (IJ), for a variety of processors in the SPARC family. *br* is the average number of branches per loop iteration. Based on Uh and Whalley.[1387]

Processor Implementation	2.5br LS	4.5br LS	8.5br LS	2.5br IJ	4.5br IJ	8.5br IJ
SPARCstation-IPC	3.82	5.53	8.82	2.61	2.71	2.76
SPARCstation-5	1.03	1.65	2.74	0.63	0.76	0.76
SPARCstation-20	0.93	1.60	2.65	0.87	0.93	0.94
UltraSPARC-1	0.50	1.16	1.56	1.50	1.51	1.51

1754 A **case** or **default** label is accessible only within the closest enclosing **switch** statement.

Commentary

This requirement needs to be explicitly stated because there is no syntactic association between **case** labels and their controlling **switch** statement.

Coding Guidelines

The issue most likely to be associated with a nested **switch** statement is source layout (because the amount of indentation used is often greater than in nested **if** statements). However, nested **switch** statements are relatively uncommon. For this reason the issue of the comprehension effort needed for this form of nested construct is not discussed.

1707 statement visual layout

1755 The integer promotions are performed on the controlling expression.

Commentary

The rationale for performing the integer promotions is the same as that for the operands within expressions.

675 integer promotions

Common Implementations

When the controlling expression is denoted by an object having a character type the possible range of values is known to fit in a byte. Even relatively simple optimizers often check for, and make use of, this special case.

1756 The constant expression in each **case** label is converted to the promoted type of the controlling expression.

Commentary

Prior to this conversion the type of the constant expression associated with each **case** label is derived from the form of the literals and result type of the operators it contains. The relationship between the value of a **case** label and a controlling expression is not the same as that between the operands of an equality operator. The conversion may cause the rank of the case label value to be reduced. If the types of both expressions are unsigned it is possible for the **case** label value to change (e.g., a modulo reduction). Like all integer conversions undefined behavior may occur for some values and types.

Other Languages

Many languages have a single integer type, so there is no conversion to perform for **case** label values. Strongly typed languages usually require that the type of the **case** label value be compatible with the type of the controlling expression, there is not usually any implicit conversions. Enumerated constants are often defined to be separate types, that are not compatible with any integer type.

Coding Guidelines

This C sentence deals with the relationship between individual **case** label values and the controlling expression. The following points deal with the relationship between different **case** label values within a given **switch** statement:

- Mixing **case** labels whose values are represented using both character constants and integer constants is making use of representation information (in this context the macro `EOF` might be interpreted in its symbolic form of representing an end-of-file character, rather than an integer constant). There does not appear to be a worthwhile benefit in having a deviation that permits the use of the integer constant 0 rather than the character constant `'\0'`, on the grounds of improved reader recognition performance. The character constant `'\0'` is the most commonly occurring character constant (10% of all character constants in the visible form of the `.c` files, even if it only represents 1% of all constant tokens denoting the value 0).
- Mixing **case** labels whose values are represented using both enumeration constants and some other form of constant representation (e.g., an integer constant) is making use of the underlying representation of the enumerated constants. The same is also true if enumerated constants from different enumerations types are mixed.
- Mixing integer constants represented using decimal, hexadecimal, or octal lexical forms. The issue of visually mixing integer constants having different lexical forms is discussed elsewhere.

form of representation mixing 1875

equality operators not floating-point operands 1214.1

Floating point literals are very rarely seen in **case** labels. The guideline recommendation dealing with exact comparison of floating-point values is applicable to this usage.

Example

```
#include <limits.h>

enum {red, green, blue};

extern int glob;

void f(unsigned char ch)
{
switch (ch)
   {
   case 'a': glob++;
             break;

   case green: glob+=2;
               break;

   case (int)7.0: glob--;
                  break;

   case 99: glob -= 9;
            break;

   case ULONG_MAX: glob *= 3;
                   break;
   }
}
```

1757 If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label.

Commentary

A **case** label can appear on any statement in the **switch** body.

```
switch (x)
   default : if (prime(x))
                case 2: case 3: case 5: case 7:
                   process_prime(x);
             else
                case 4: case 6: case 8: case 10:
                   process_composite(x);
```

There can be more practical uses for this functionality. 1766 Duff's Device

Coding Guidelines

Experience suggests that developers treat the case label value as being the result of evaluating the expression appearing in the source (i.e., that no conversion, driven by the type of the controlling expression, takes place). A conversion that causes a change of value is very suspicious. However, no instances of such an event occur in the Usage `.c` files or have been experienced by your author. Given this apparent rarity no guideline recommendation is made here.

1758 Otherwise, if there is a **default** label, control jumps to the labeled statement.

Commentary

A **switch** statement may be thought of as a series of **if** statements with the **default** label representing the final **else** arm (although other **case** labels may label the same statement as a **default** label).

Common Implementations

Having a **default** label may not alter the execution time performance of the generated machine code. All of the tests necessary to determine that the default label should be jumped to are the same as those necessary to determine that no part of the switch should be executed (if there is no **default** label).

1759 If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

Commentary

This behavior is the same as that of a series of nested **if** statements. If all of their controlling expressions are false and there is no final **else** arm, none of the statement bodies is executed.

Other Languages

Some languages require that there exist a **case** label value, or **default**, that matches the value of the controlling expression. If there is no such matching value the behavior may be undefined (e.g., Pascal specifies it is a *dynamic-violation*) or even defined to raise an exception (e.g., Ada).

Coding Guidelines

The coding guideline issue of always having a **default** label is discussed elsewhere. 1751 default label at most one

Implementation limits

1760 As discussed in 5.2.4.1, the implementation may limit the number of **case** values in a **switch** statement.

Commentary

This observation ought really to be a Further reference subclause. 296 limit case labels

footnote 133

133) That is, the declaration either precedes the **switch** statement, or it follows the last **case** or **default** label associated with the **switch** that is in the block containing the declaration. 1761

Commentary

If the declaration is not followed by any **case** or **default** labels, all references to the identifier it declares can only occur in the statements that follow it (which can only be reached via a jump to preceding **case** or **default** labels, unless a **goto** statement jumps to an ordinary label within the statement list occurs).

EXAMPLE case fall through

EXAMPLE In the artificial program fragment 1762

```
switch (expr)
{
        int i = 4;
        f(i);
case 0:
        i = 17;
        /* falls through into default code */
default:
        printf("%d\n", i);
}
```

the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

Commentary

static storage duration 151
initialized before startup

Objects with static storage duration are initialized on program startup.

```
1  switch (i)
2     {
3     static char message[] = "abc"; /* Not dependent on control flow. */
4     case 0:
5          f(message);
6          break;
7     case 1:
8          /* ... */
9     }
```

case 1727 fall through
switch 1749 past variably modified type

Other issues associated with constructs contained in this example are discussed elsewhere.

6.8.5 Iteration statements

iteration statement syntax

1763

iteration-statement:

while **(** *expression* **)** *statement*
do *statement* **while** **(** *expression* **)** **;**
for **(** $expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for **(** *declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

Commentary

The terms *loop header* or *head of the loop* are sometimes used to refer to the source code location containing the controlling expression of a loop (in the case of a **for** statement it might be applied to all three components bracketed by parentheses).

It is often claimed that programs spend 90% of their time executing 10% of their code. This characteristic is only possible if the time is spent in a subset of the programs iteration statements, or a small number of functions called within those statements. While there is a large body of published research on program

performance, there is little evidence to back up this claim (one study[1316] found that 88% of the time was spent in 20% of the code, while analysis[1422] of some small embedded applications found that 90% of the time was spent in loops). It may be that researchers are attracted to applications which spend their time in loops because there are often opportunities for optimization. Most of existing, published, execution time measurements are based on engineering and scientific applications, for database oriented applications[1141] and operating systems[1358] loops have not been found to be so important.

The **;** specified as the last token of a **do** statement is not needed to reduce the difficulty of parsing C source. It is simply part of an adopted convention.

C90

Support for the form:

for (*declaration* $expr_{opt}$ **;** $expr_{opt}$ **)** *statement*

is new in C99.

C++

The C++ Standard allows local variable declarations to appear within all conditional expressions. These can occur in **if**, **while**, and **switch** statements.

Other Languages

Many languages require that the lower and upper bounds of a **for** statement be specified, rather than a termination condition. They usually use keywords to indicate the function of the various expressions (e.g., Modula-2, Pascal):

```
FOR I=start TO end BY step
```

Some languages (e.g., BCPL, Modula-2) require `step` to be a translation time constant. Both Ada or Pascal require **for** statements to have a step size of one. Ada uses the syntax:

```
for counter in 1..10
   loop
   ...
for counter in reverse 1..10
   loop
   ...
```

which also acts as the definition of `counter`.

Cobol supports a **PERFORM** statement, which is effectively a **while** statement.

```
        PERFORM UNTIL quantity > 1000
       * some code
        END-PERFORM
```

The equivalent looping constructs In Fortran is known as a **do** statement. A relatively new looping construct, at least in the Fortran Standard, is **FORALL**. This is used to express a looping computation in a form that can more easily be translated for parallel execution. Some languages (e.g., Modula-2, Pascal) use the keywords **repeat/until** instead of **do/while**, while other languages (e.g., Ada) do not support an iteration statement with a test at the end of the loop.

A few languages (e.g., Icon[234] which uses the term *generators*) have generalized the looping construct to provide what are commonly known as *iterators*. An iterator enumerates the members of a set (a mechanism for accessing each enumerated member is provided in the language), usually in some unspecified order, and has a loop termination condition.

Common Implementations

Many programs spend a significant percentage of their time executing iteration statements. The following are some of the ways in which processor and translator vendors have responded to this common usage characteristic:

Measuring implementations 0

translator optimizations 0

loop unrolling 1774

data dependency 988

cache 0

translator performance 0
vs. assembler
DSP processors 0

- Translator vendors wanting to optimize the quality of generated machine code have a number of optimization techniques available to them. A traditional loop optimization is strength reduction[277] (which replaces costly computations by less expensive ones), while more ambitious optimizers might perform hoisting of loop invariants and loop unrolling. Loop invariants are expressions whose value does not vary during the iteration of a loop; such expressions can be hoisted to a point just outside the start of the loop. Traditionally translators have only performed loop unrolling on **for** statements. (Translation time information on the number of loop iterations and step size is required; this information can often be obtained by from the expressions in the loop header, i.e., the loop body does not need to be analyzed.) More sophisticated optimizations include making use of data dependencies to order the accesses to storage. As might be expected with such a performance critical construct, a large number of other optimization techniques are also available.
- Processor vendors want to design processors that will execute programs as quickly as possible. Holding the executed instructions in a processor's cache saves the overhead of fetching them from storage and most processors cache both instructions and object values. Some processors (usually DSP) have what is known as a *zero overhead loop buffer* (effectively a software controlled instruction cache). The sequence of instructions in such a loop buffer can be repetitively executed with zero loop overhead (the total loop count may be encoded in the looping instruction or be contained in a register). Because of their small size (the Agere DSP16000[6] loop buffer has a limit of 31 instructions) and restrictions on instructions that may be executed (e.g., no instructions that change the flow of control) optimizers can have difficulty making good of such buffers.[1386] The characteristics of loop usage often means that successive array elements are accessed on successive loop interactions (i.e., storage accesses have spatial locality). McKinley and Temam[918] give empirical results on the effect of loops on cache behavior (based on Fortran source). Some CISC processors support a decrement/increment and branch on nonzero instruction;[319,615] ideal for implementing loops whose termination condition is the value zero (something that can be arranged in handwritten assembler, but which rarely happens in loops written in higher-level languages— Table 1763.1). The simplifications introduced by the RISC design philosophy did away with this kind of instruction; programs written in high-level languages did not contain enough loops of the right kind to make it cost effective supporting such an instruction. However, one application domain where a significant amount of code is still written in assembler (because of the comparatively poor performance of translator generated machine code) is that addressed by DSP processors, which often contain such decrement (and/or increment) branch instructions (the SC140 DSP core[972] includes hardware loop counters that support up to four levels of loop nesting). The C compiler for the Unisys e-@ction Application Development Solutions[1391] uses the `JGD` processor instruction to optimize the loop iteration test. However, this usage limits the maximum number of loop iterations to $2^{35} - 2$, a value that is very unlikely to be reached in a commercial program (a trade-off made by the compiler implementors between simplicity and investing effort to handle very rare situations).

Obtaining an estimate of the execution time of a sequence of statements may require estimating the number of times an iteration statement will iterate. Some implementations provide a mechanism for the developer to provide iteration count information to the translator. For instance, the translator for the TMS320C6000[1343] supports the following usage:

```
1   #pragma MUST_ITERATE (30) /* Will loop at least 30 times. */
```

Another approach is for the translator to deduce the information from the source.[557]

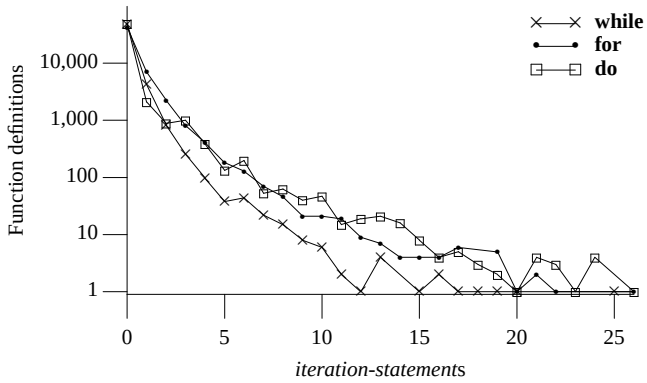


Figure 1763.1: Number of function definitions containing a given number of *iteration-statement*s. Based on the translated form of this book's benchmark programs.

Program loops may not always be expressed using an *iteration-statement* (for instance, they may be created using a **goto** statement). Ramalingam[1140] gives an algorithm for identifying loops in almost linear time.

Example

```
#include <stdio.h>

int f(unsigned char i, unsigned char j)
{
do
while (i++ < j)
   ;
while (i   > j++)
   ;

if (j != 0)
   printf("Initial value of i was greater than initial value of j\n");
}
```

Usage

A study by Bodík, Gupta, and Soffa[131] found that 11.3% of the expressions in SPEC95 were loop invariant.

Table 1763.1: Occurrence of various kinds of **for** statement controlling expressions (as a percentage of all such expressions). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of **for** loop header	%
assignment ; identifier < identifier ; identifier **v++**	33.2
assignment ; identifier < *integer-constant* ; identifier **v++**	11.3
assignment ; identifier ; assignment	7.0
assignment ; identifier < expression ; identifier **v++**	3.3
assignment ; identifier < identifier ; **++v** identifier	2.7
; ;	2.5
assignment ; identifier != identifier ; assignment	2.5
assignment ; identifier <= identifier ; identifier **v++**	2.2
assignment ; identifier >= *integer-constant* ; identifier **v--**	1.6
assignment ; identifier < function-call ; identifier **v++**	1.4
assignment ; identifier < identifier ; identifier **v++** , identifier **v++**	1.4
others	31.1

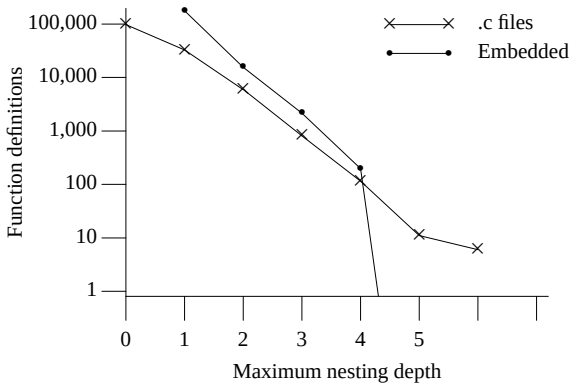


Figure 1763.2: Number of functions containing *iteration-statement*s nested to the given maximum nesting level; for embedded C[389] (whose data was multiplied by a constant to allow comparison) and the visible form of the `.c` files (zero nesting depth denotes functions not containing any *iteration-statement*s).

Table 1763.2: Occurrence of various kinds of **while** statement controlling expressions (as a percentage of all **while** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	43.5	expression	2.2
object	12.2	`*v` object	2.0
object `!=` object	7.0	assignment	1.8
integer-constant	6.2	`!` object	1.6
object `<` object	4.7	`!` function-call	1.3
function-call	4.4	object `!=` *integer-constant*	1.2
object `>` *integer-constant*	4.0	object `v--` `>` *integer-constant*	1.1
object `v--`	3.2	`!` expression	1.0
assignment `!=` object	2.4		

Constraints

controlling expression iteration statement

The controlling expression of an iteration statement shall have scalar type. 1764

Commentary

if statement 1743 controlling expression scalar type

The issues here are the same as for controlling expressions in **if** statements.

Other Languages

Many languages do not support loop control variables having a pointer type (invariably because they do not support any form of pointer arithmetic).

Coding Guidelines

loop control variable 1774

The concept of *loop control variable* is discussed elsewhere.

for statement declaration part

The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**. 1765

Commentary

loop control variable 1774

The intent is to support the declaration of an identifier that is used as a loop control variable. There are many people who believe that limiting the scope over which such control variables can be modified is a *good thing*. Another coding guideline related issue is that of declaration of identifiers occurring close to where they are used in statements (this issue is discussed elsewhere).

declaration 1348 syntax
statement 1707 syntax

Problem

Consider the code:

```
for (enum fred { jim, sheila = 10 } i = jim; i < sheila; i++)
     // loop body
```

Proposed Committee Response

The intent is clear enough; `fred`*,* `jim`*, and* `sheila` *are all identifiers which do not denote objects with* **`auto`** *or* **`register`** *storage classes, and are not allowed in this context.*

C90

Support for this functionality is new in C99.

C++

The `declarator` *shall not specify a function or an array. The* `type-specifier-seq` *shall not contain* **`typedef`** *and shall not declare a new class or enumeration.* 6.4p2

```
1  void f(void)
2  {
3  for (int la[10]; /* does not change the conformance status of the program */
4                   // ill-formed
5       ; ;)
6     ;
7  for (enum {E1, E2} le; /* does not change the conformance status of the program */
8                          // ill-formed
9       ; ;)
10    ;
11 for (static int ls; /* constraint violation */
12                     // does not change the conformance status of the program
13      ; ;)
14    ;
15 }
```

Other Languages

In some languages (e.g., Ada and Algol 68) the identifier used as a loop control variable in a **`for`** statement is implicitly declared to have the appropriate type (based on the type of the expressions denoting the start and end values).

Coding Guidelines

The ability to declare identifiers in this context is new in C99 and at the time of this writing there is insufficient experience with its use to know whether any guideline recommendation is worthwhile.

Semantics

1766 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0.

iteration statement executed repeatedly loop body

Commentary

This defines the term *loop body*. The term *loop* is commonly used as a noun by developers to refer to constructs associated with iteration statements (which are rarely referred to as *iteration statements* by developers). For instance, the terms *loop statement*, or simply a *loop* are commonly used by developers.

if statement 1744 operand compare against 0

Execution of the loop may also terminate because a **break**, **goto**, or **return** statement is executed. The discussion on the evaluation of the controlling expression in an **if** statement is applicable here.

Duff's Device

It is often necessary to access a block of storage (e.g., to copy it somewhere else, or to calculate a checksum of its contents). For anything other than the smallest of blocks the overhead of a loop can be significant.

```
1  void send(register unsigned char *to,
2            register unsigned char *from,
3            register int count)
4  {
5  do
6      *to++ = *from++;
7  while (--count > 0);
8  }
```

The above loop requires a comparison after ever item copied. Unrolling the loop would reduce the number of comparisons per items copied. However, because `count` is not known at translation time an optimizer is unlikely to perform loop unrolling. The loop can be unrolled by hand, making sure that code also handles the situation where the number of items being copied is not an exact multiple of the loop unroll factor. A technique proposed by Tom Duff[371] (usually referred to as *Duff's device*) is (the original example used `*to`, i.e., the bytes were copied to some memory mapped serial device):

```
1  void send(register unsigned char *to,
2            register unsigned char *from,
3            int count)
4  {
5  register int n = (count+7)/8;
6
7  switch (count % 8)
8     {
9     case 0: do{   *to++ = *from++;
10    case 7:       *to++ = *from++;
11    case 6:       *to++ = *from++;
12    case 5:       *to++ = *from++;
13    case 4:       *to++ = *from++;
14    case 3:       *to++ = *from++;
15    case 2:       *to++ = *from++;
16    case 1:       *to++ = *from++;
17                } while (--n > 0);
18    }
19 }
```

C++

The C++ Standard converts the controlling expression to type **bool** and expresses the termination condition in terms of true and false. The final effect is the same as in C.

Other Languages

In many other languages the model of a *for loop* involves a counter being incremented (or decremented) from a start value to an end value, while the model of a *while loop* (or whatever it is called) being something that iterates until some condition is met. There is considerable overlap between these two models (it is always possible to rewrite one form of loop in terms of the other). The differences between the two kinds of loop are purely conceptual ones, created by developer loop classification models. Loop classification is often based on deciding whether a loop has the attributes needed to be considered a *for loop* (e.g., the number of iterations being known before the first iteration starts), all other loops being classified as *while loop*s. Early versions of Fortran performed the loop termination test at the end of the loop. This meant that loops always iterated at least once, even if the test was false on the first iteration.

Coding Guidelines

Some coding guideline documents recommend that loop termination only occur when the condition expressed in the controlling expression becomes equal to zero. A number of benefits are claimed to accrue from adhering to this recommendation. These include, readers being able to quickly find out the conditions under which the loop terminates (by looking at the loops controlling expression; which might only be a benefit for one form of reading) and the desire not to jump across the control flow. It is always possible to transform source code into a form where loop termination is decided purely by the control expression. However, is there a worthwhile cost/benefit to requiring such usage? The following example illustrates a commonly seen need to terminate a loop early:

770 reading kinds of
1782 jump statement syntax

```
#define SPECIAL_VAL 999
#define NUM_ELEMS 10

extern int glob;
static int arr[NUM_ELEMS];

void f_1(void)
{
for (int loop = 0; loop < NUM_ELEMS; loop++)
   {
   /* ... */
   if (glob < NUM_ELEMS/2)
      {
      glob++;
      if (arr[loop] == SPECIAL_VAL)
         break;
      }
   else
      arr[loop] = glob;
   /* ... */
   }
}

void f_2(void)
{
for (int loop = 0; loop < NUM_ELEMS; loop++)
   {
   /* ... */
   if (glob < NUM_ELEMS/2)
      {
      glob++;
      if (arr[loop] == SPECIAL_VAL)
         loop = NUM_ELEMS;
      }
   else
      arr[loop] = glob;

   if (loop != NUM_ELEMS)
      { /* ... */ }
   }
}

void f_3(void)
{
_Bool terminate_early = 0;

for (int loop = 0; (loop < NUM_ELEMS) && !terminate_early; loop++)
   {
   /* ... */
   if (glob < NUM_ELEMS/2)
      {
```

```
52          glob++;
53          if (arr[loop] == SPECIAL_VAL)
54             terminate_early = 1;
55          }
56       else
57          arr[loop] = glob;
58
59       if (!terminate_early)
60          { /* ... */ }
61       }
62    }
```

Looking at the controlling expression in `f_1` and `f_2` it appears to be easy to deduce the condition under which the loop will terminate. However, in both cases the body of the loop contains a test that also effectively terminates the loop (in the case of `f_2` the body of the loop has increased in complexity by the introduction of an **`if`** statement). The function `f_3` handles the case where guidelines recommend against modifying the loop control variable in the loop body.

loop control variable 1774

Any guideline recommendation needs to be based on a comparison of the costs and benefits of the loop constructs in these functions (and other cases). Your author knows of no studies that provide the information needed to make such a comparison. For this reason this coding guideline subsection is silent on the issue of how loops might terminate. A loop where it is known, at translation time, that the number of iterations is zero, is a loop containing redundant code. The issue of redundant code is discussed elsewhere.

redundant code 190

The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[DR268] 1767

Commentary

This is a requirement on the implementation.

This sentence was added by the response to DR #268.

Other Languages

Many languages (e.g., Pascal, Ada) treat loop bodies as indivisible entities and do not permit a jump into them (although it is usually possible to jump out of them).

Coding Guidelines

Some coding guideline documents recommend against jumping into the body of a loop. One argument is that a reader of the source may not notice that a loop could be entered in this way and makes a modification that fails to take this case into account (i.e., introduces a fault).

There are a variety of situations where jumping into the body of a loop may result in code that is less likely to contain faults and be less costly to maintain (see the example given for the **`goto`** statement).

goto EXAMPLE 1790

Jumping into the body of a loop is rare and no data is available on the kinds of faults in which it plays a significant contributing factor. For this reason a no guideline recommending is given.

---

block iteration statement

An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. 1768

Commentary

The rationale for this specification is the same as that given for the block implicitly created for a selection statement.

block selection statement 1741

---

block loop body

The loop body is also a block whose scope is a strict subset of the scope of the iteration statement. 1769

Commentary

The rationale for this specification is the same as that given for the block implicitly created for the substatements of a selection statement.

block selection substatement 1742

Example

In the following example the lifetime of the compound literal starts and terminates on every iteration of the loop.

```
1   struct S {
2             int mem_1;
3             int mem_2;
4             };
5
6   extern void g(struct S);
7
8   void f(void)
9   {
10  for (int i = 0; i < 10; i++)
11     g((struct S){.mem_1 = i, .mem_2 = 42});
12  }
```

1770 DR268) Code jumped over is not executed.

Commentary

This is a requirement on the implementation and is consistent with other situations where code is jumped over.

1762 EXAMPLE case fall through

This sentence was added by the response to DR #268.

1771 In particular, the controlling expression of a **for** or **while** statement is not evaluated before entering the loop body, nor is *clause-1* of a **for** statement.

Commentary

While any expressions in the loop header are not executed when the loop body is entered, the controlling expression evaluation that occurs at start of the next, and any subsequent, iterations of the loop is executed.

1763 iteration statement syntax
1766 iteration statement executed repeatedly

This sentence was added by the response to DR #268.

6.8.5.1 The while statement

1772 The evaluation of the controlling expression takes place before each execution of the loop body.

while statement

Commentary

The loop body of a **while** statement may be executed zero or more times.

Coding Guidelines

Why do developers choose to use a **while** statement rather than a **for** statement? Technically a loop can be written using either kind of statement. Both forms of iteration statement are likely to involve initializing, testing, and modifying one or more objects that systematically change over successive iterations. The **for** statement places these three components in a contiguous, visibly prominent, location. Other reasons for the choice (65.5% **for**, 34.5% **while**) include:

- *C culture.* The use of a particular kind of loop to perform a particular operation may be something that developers learn as part of the process of becoming a C programmer. Measurements of the two looping constructs (see Table 1763.1 and Table 1763.2) show that **for** statements often count up to some value and **while** statements iterate until an equality operator is true. The pattern of usage seen in the source being the sum of the operations (e.g., always using a **for** statement to loop over the elements of an array and a **while** statement to loop over a linked list) required to implement the application.

- *Individual habits.* While learning to program a developer may have chosen (perhaps a random selection, or purely a class exercise to practice the using a language construct) to use a particular construction to perform some operation. Reuse of the same construction to perform the same, or similar operations leads to it becoming established as part of their repertoire. The pattern of usage seen in source code being the sum of individual habits.

In both cases the choice of **for**/**while** involves a process of algorithmic problem classification. Which most closely matches the developers mental model of the operations being performed? At the time of this writing there is insufficient information to evaluate whether there is a cost/benefit case to the use of **while** statements, rather than **for** statements. These coding guidelines do not discuss this issue any further.

loop control variable 1774 controlling expression 1740 if statement

There is a commonly seen idiom that uses side effects in the evaluation of the controlling expression to modify the value of an object in the controlling expression (i.e., the loop control variable). The discussion on controlling expressions in an **if** statement showed that removing such side effects in the controlling expression of **while** statements would incur the cost of having to create and maintain two identical statements (one outside the loop and one inside). Your author is not able to estimate if this cost was less than the potential benefits of not having the side effects in the controlling expression. For this reason no guideline is specified here.

6.8.5.2 The do statement

do statement

The evaluation of the controlling expression takes place after each execution of the loop body. 1773

Commentary

The loop body of a **do** statement is always executed at least once.

Coding Guidelines

while statement 1772

The benefits associated with having side effects in the controlling expression of a **while** statement are not applicable to a **do** statement (because the loop is always executed at least once). Given that the use of a **do** statement is relatively rare and that developers are likely to be familiar with the side effect idioms that occur in controlling expressions, no guideline recommendation is given here.

Example

macro function-like 1933

One use of the **do** statement is to solve the dangling semicolon problem that can occur when a function-like macro replaces a function call. Bracketing a sequence of statement with braces creates a compound statement, which does not require a terminating semicolon. In most contexts a semicolon following a function-like macro invocation is a harmless null statement. However, as the following example shows, when it forms the first arm of an **if** statement that contains an **else** arm, the presence of a semicolon is a syntax violation. Enclosing a sequence of statements in the body of a **do** statement, whose controlling expression is false, avoids this problem.

```
#define STMT_SEQ(p)                  \
      do {                           \
         /* sequence of statements */ \
         }                           \
      while (0)

/* ... */

if (cond)
   STMT_SEQ(x);
else
   /* ... */
```

6.8.5.3 The for statement

1774 The statement

for statement

```
for ( clause-1 ; expression-2 ; expression-3 ) statement
```

behaves as follows:

Commentary

Rationale

In C89, for loops were defined in terms of a syntactic rewrite into **while** loops. This introduced problems for the definition of the **continue** statement; and it also introduced problems when the operands of cast and **sizeof** operators contain declarations as in:

```
enum {a, b};
{
    int i, j = b;
    for (i = a; i < j; i += sizeof(enum {b, a}))
        j += b;
}
```

not being equivalent to:

```
enum {a, b};
{
    int i, j = b;
    i = a;
    while (i < j) {
        j += b;                     // which b?
        i += sizeof(enum {b, a});  // declaration of b moves
    }
}
```

because a different `b` is used to increment `i` in each case. For this reason, the syntactic rewrite has been replaced by words that describe the behavior.

C90

Except for the behavior of a **continue** *statement in the loop body, the statement*

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

and the sequence of statements

```
expression-1 ;
while (expression-2) {
statement ;
expression-3 ;
}
```

are equivalent.

C++

Like the C90 Standard, the C++ Standard specifies the semantics in terms of an equivalent **while** statement. However, the C++ Standard uses more exact wording, avoiding the possible ambiguities present in the C90 wording.

Other Languages

loop control variable 1774

In most other languages the ordering of expressions puts the controlling expression last. Or to be more exact, an upper or lower bound for the loop control variable appears last. Most other languages do not support having anything other than the loop control variable tested against a value that is known at translation time. Some languages (e.g., Ada, Algol 68, and Pascal) do not allow the loop control variable to be modified by the body of the loop.

Common Implementations

loop unrolling

Loop unrolling is the process of decreasing the number of iterations a loop makes by duplicating the statement in the loop body.[324] For instance:

```
1   for (loop = 0; loop < 10; loop++)
2      {
3      a[loop] = loop;
4      }
5   /*
6    * Can be unrolled to the following equivalent form:
7    */
8   for (loop = 0; loop < 10; loop+=2)
9      {
10     a[loop] = loop;
11     a[loop+1] = loop+1;
12     }
```

Loop unrolling reduces the number of jumps performed (which can be a significant saving when the loop body is short) and by increasing the number of statement in the loop body creates optimization opportunities (which, in the above example, could result in two loop bodies executing in less time than twice the time for a single iteration). When the iteration count is not exactly divisible by the loop body unrolling factor copies of the loop body may need to occur before the start, or after the end, or the loop statement.

At the minimum, loop unrolling requires knowing the number of loop iterations and the amount by which the loop control variable is incremented, at translation time. Implementations often place further restrictions on loops before that they unroll (requiring the loop body to consist of a single basic block is a common restriction).

Arbitrary amounts of loop unrolling (e.g., iterating 10 times over 100 copies of a loop body where the original is known to iterate 1000 times) does not necessarily guarantee improved performance. Duplicating the loop body increases code size, which decreases the probability that all of the loop body instructions will fit within the processor's instruction cache. Unless optimizers take into account the size of a processor's instruction cache when evaluating the cost effectiveness of loop unrolling they can end up reducing, rather than increasing, program performance.[484]

iteration statement syntax 1763

Jinturkar[670] analyzed the loops in a set of benchmarks to determine the complexity and size of loop bodies, and the nature of the loop bounds. The results were that in 50% of loops the iteration count could be deduced at translation time and that the generated machine code for the loop bodies of each loop that were unrolled was smaller than 256 bytes.

Coding Guidelines

Writers of coding guideline documents often regard the components of a **for** statement as having attributes that other loop statements don't have (e.g., they have an associated loop control variable). While it can be argued that many of these authors have simply grafted onto C concepts that only exist in other languages (or perhaps the encapsulation all of the loop control information in one visually delimited area of the source

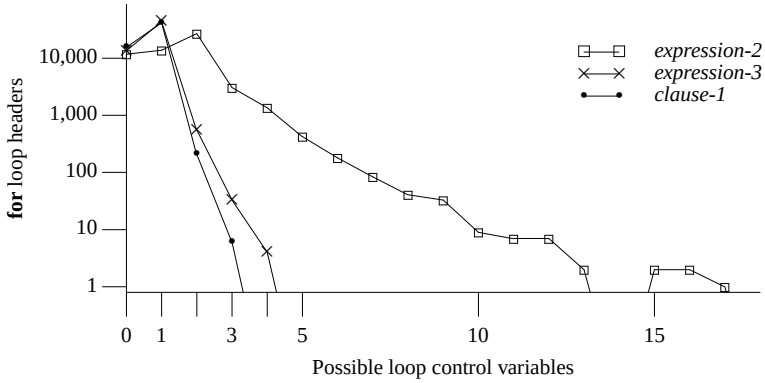


Figure 1774.1: Number of possible loop control variables appearing in *expression-2* (square-box) after filtering against the objects appearing in *expression-3* (cross) and after filtering against the objects appearing in *clause-1* (bullet). Based on the visible form of the `.c` files.

triggers a cognitive response that triggers implicit assumptions in readers), if a sufficient number of developers associate these attributes with **for** statements then they become part of the culture of C and need to be considered here. Other loop conceptualization issues are discussed elsewhere.

1772 while statement

This subsection discusses one attribute commonly associated with **for** statements that is not defined by the C Standard, the so-called *loop control variable* (or simply *loop variable*, or alternatively *loop counter*). A loop control variable is more than simply a concept that might occur during developer discussion, many coding guideline documents make recommendations about its use (e.g., a loop control variable should not be modified during execution of the body of the loop, or have floating-point type). Which of the potentially four different objects that might occur, for instance, in the most common form of loop header (see Table 1763.1) is the loop control variable?

loop control variable

0 MISRA

```
1   for (lcv_1=0; lcv_2 < lcv_3; lcv_4++)
```

The following algorithm frequently returns an answer that has been found to be acceptable to developers (it is based on the previous standard and has not been updated to reflect the potential importance of objects declared in *clause-1*). Note that the algorithm may return zero, or multiple answers; a union or structure member selection operator and its two operands is treated as a single object, but both an array and any objects in its subscript are treated as separate objects and therefore possible loop control variables:

1. list all objects appearing in *expression-2* (the controlling expression). If this contains a single object, it is the loop control variable (33.2% of cases in the `.c` files),
2. remove all objects that do not appear in *expression-3* (which is evaluated on every loop iteration). If a single object remains, that is the loop control variable (91.8% of cases in the `.c` files),
3. remove all objects that do not appear in *clause-1* (which is only evaluated once, prior to loop iteration). If a single object remains, that is the loop control variable (86.2% of cases in the `.c` files).

Unlike the example given above, in practice the same object often appears as an operand somewhere within all three components (see Figure 1774.1).

Because the controlling expression is evaluated on every iteration of the loop, the loop control variable can appear in contexts that are not supported in other languages (because most evaluate the three loop components only once, prior to the first iteration). For instance:

```
1   for (lcv_1=0, lcv_2=0; a1[lcv_1] < a2[lcv_2]; lcv_1++, lcv_2+=2)
```

Experience shows that developers often assume that, in a **for** statement, modification of any loop control variables only occurs within the loop header. This leads to them forming beliefs about properties of the loop, for instance, *it loops 10 times*. There tend to be fewer assumptions made about the use of **while** statements (which might not even be thought to have a loop control variable associated with them) and the following guideline is likely to cause developers to use this form of looping construct.

Cg 1774.1 A loop control variable shall not be modified during the execution of the body of a **for** statement.

Some coding guideline documents recommend that loop control variables not have floating-point type. It might be thought that such a recommendation only makes sense in languages where the loop termination condition involves an equality test (in C this case is covered by the guideline recommendation dealing with the type of the operands of the quality operators). However, the controlling expression in a C **for** statement can contain relational operators, which can also have a dependence on the accuracy of floating-point operations. For instance, it is likely that the author of the following fragment expects the loop to iterate 10 times. However, it is possible that 10 increments of `i` result in it having the value 9.9999, and loop termination not occurring until after the eleventh iteration.

equality operators not floating-point operands 1214.1

```
1   for (float i=0.0; i < 10.0; i++)
```

A possible developer response to a guideline recommendation that loop control variables not have floating point type is to use a **while** statement (which are not covered by the algorithm for deducing loop control variables). Some of the issues associated with the finite accuracy of operations on floating-point values can be addressed with guideline recommendations. However, the difficulty of creating wording for a recommendation dealing with the use of floating-point values to control the number of loop iterations is such that none is attempted here.

Table 1774.1: Occurrence of sequences of components omitted from a **for** statement header (as a percentage of all **for** statements). Based on the visible form of the `.c` files.

Components Omitted	%
clause-1	3.8
clause-1 expr-2	0.1
clause-1 expr-2 expr-3	2.5
clause-1 expr-3	0.1
expr-2	0.8
expr-2 expr-3	0.2
expr-3	1.6

controlling expression for statement

The expression *>expression-2* is the controlling expression that is evaluated before each execution of the loop body. 1775

Commentary

The loop body of a **for** statement may be executed zero or more times.

Other Languages

In many languages the termination condition of a **for** statement, specified in the header of the loop body, is fixed prior to the first iteration of the loop body (every time the loop statement is encountered). The commonly used termination condition being that the loop counter be equal to the value of a specified expression.

Coding Guidelines

The controlling expression in a **for** statement is sometimes written so that its evaluation also has the side effect of modifying the value of the loop control variable, removing the need for *expression-3*. A developer

may have any of a number of reasons for using such an expression, from use of an idiom to misplaced concern for efficiency (many of the issues associated with side effects within the controlling expression are the same as those that apply to **while** statements). 1772 while statement

---

1776 The expression *expression-3* is evaluated as a void expression after each execution of the loop body.

Commentary

The common usage is for the evaluation of this expression to modify the value of the loop control variable.

Other Languages

In many languages the value used to increment/decrement the loop counter of a **for** statement is fixed (every time the **for** statement is encountered) prior to the first iteration of the loop body.

1777 If *clause-1* is a declaration, the scope of any ~~variables~~ identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions;

Commentary

The phrase *entire loop* means *expression-2*, *expression-3*, and the loop body. 1766 loop body

The wording was changed by the response to DR #292.

C90

Support for this functionality is new in C99.

Other Languages

In some languages (e.g., Ada and Algol 68) the occurrence of an identifier as the loop control variable also acts as a definition of that identifier (its type is that of the controlling expressions).

Coding Guidelines

Some coding guideline documents warn of the dangers of accessing loop control variables outside of the loops they control. One of the reasons for this is that some languages do not define the value of this variable once the loop has terminated (which makes such accesses equivalent to those of an uninitialized variable). Loop control variables in C do not have this behavior and some form of related guideline recommendation is not required.

Declaring the loop control variable via *clause-1* has the benefit of localizing the visual context over which it is referenced. Possible costs include having to modify existing habits (e.g., looking for the declaration at the start of the function body) and possible lack of support for constructs new in C99 by ancillary tools. 1768 block iteration statement

Example

```
1  void three_different_objects_named_loc(void)
2  {
3  int loc = 20;
4
5  for (int loc = 0; loc < 10; loc++)
6     int loc = -8;
7  }
```

1778 it is reached in the order of execution before the first evaluation of the controlling expression.

Commentary

This requirement is necessary because it is intended that objects declared in *clause-1* appear in the controlling expression.

If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[134)] 1779

Commentary

This evaluation occurs once, every time the **for** statement head of the loop is encountered in the flow of control.

Both *clause-1* and *expression-3* can be omitted. 1780

Commentary

Saying in words what is specified in the syntax. A **for** statement loop header is essentially a means of visually highlighting the various components of a loop.

C++

The C++ Standard does not make this observation, that can be deduced from the syntax.

Other Languages

Being able to omit the specification for the initial value of a loop counter (i.e., *clause-1*) is unique to C (and C++). Most languages allow their equivalent of *expression-3* to be omitted and use a default value (usually either 1 or -1).

Coding Guidelines

Why would a developer choose to omit either of these constructs in a **for** statement, rather than using a **while** statement? This issue is discussed elsewhere.

iteration statement 1763
syntax
while statement 1772

An omitted *expression-2* is replaced by a nonzero constant. 1781

Commentary

Specifying that an omitted *expression-2* is replaced by a nonzero constant allows a more useful meaning to be given to those cases where *clause-1* or *expression-3* are present, than by replacing it by the constant 0. Omitting *expression-2* creates a loop that can never terminate via a condition in the loop header. Executing a **break**, **goto**, or **return** statement (or a call to the `longjmp` library function) can cause execution of the loop to terminate. The term *infinite loop* is often used to describe a **for** statement where the controlling expression has been omitted. In some freestanding environments the main body of a program consists of an infinite loop that is only terminated when electrical power to the processor is switched off.

Other Languages

Most languages require that the loop termination condition be explicitly specified. In Ada the loop header is optional (a missing header implies an infinite loop).

Common Implementations

The standard describes an effect that most implementations do not implement as stated. A comparison that is unconditionally true can be replaced by an unconditional jump.

Coding Guidelines

There is an idiom that omits *expression-2* when an infinite loop is intended (usually omitting the other two expressions as well). Does such an idiom have a more worthwhile cost/benefit than using a **while** statement, with a nonzero constant as the controlling expression? Existing source code contains both usages (see Table 1763.1, Table 1763.2) and given practice readers will learn to automatically recognize both forms. However, such automatic recognition takes time to learn and a **while** statement whose controlling expression is a nonzero constant probably requires less effort to comprehend (because it is not an implicit special case) for less experienced developers.

Example

```
#define TRUE 1

void f(void)
{
for (;;)
   { /* ... */ }

while (TRUE)
   { /* ... */ }
}
```

6.8.6 Jump statements

jump statement syntax

1782

jump-statement:
goto *identifier* **;**
continue ;
break ;
return $expression_{opt}$ **;**

Commentary

These are all jump statements in the sense they cause the flow of control to jump to another statement (in the case of the **goto** statement this could be itself).

Other Languages

Most imperative languages contain some form of **goto** statement, even those intended for applications involving safety-critical situations. Snobol 4 does not have an explicit jump statement. All statements may be followed by the name of a label, which is jumped to (it can be conditional on the success or failure of the statement) on completion of execution of the statement. The **come from** statement is described by Clark.[246]

A number of languages support some mechanism for early loop termination (e.g., Ada and Modula-2 support an **exit** statement). Some languages require the exit point to be labeled, others simply exit the loop containing the statement. Perl uses the keywords **next** and **last**, rather than **continue** and **break** respectively.

Common Implementations

On most modern processors instruction execution is broken down into stages that are executed in sequence (known as an *instruction pipeline*). Optimal performance requires that this pipeline be kept filled with instructions. Jump statements (or rather the machine code generated to implement them) disrupt the smooth flow of instructions into the pipeline. This disruption occurs because the instruction fetch unit assumes the next instruction executed will the one following the current instruction, the processor is not aware it has encountered a branch instruction until that instruction has been decoded, by which time it is one or more stages down the pipeline and the following instruction is already in the pipeline behind it. Until the processor executes the branch instruction it does not know which location to fetch the next instruction from, a pipeline *stall* has occurred. Branch instructions are relatively common, which means that pipeline stalls can have a significant performance impact. The main techniques used by processor vendors to reduce the impact of stalls are discussed in the following C sentences.

0 processor pipeline

One of the design principles of RISC was to expose some of the underlying processor details to the translator, in the hope that translators would make use of this information to improve the performance of the generated machine code. Some of the execution delays causes by branch instructions have been

exposed. For instance, many RISC processors have what is known as a *delay slot* immediately after a branch instruction. The instruction in this delay slot is always executed before the jump occurs (some processors allow delay slot instructions following a conditional branch to be annulled). This delay slot simplifies the processor by moving some of the responsibility for keeping the pipeline full to the translator writer (who at worst fills it with a no-op instruction). Most processors have a single delay slot, but the Texas Instruments TMS320C6000[1342] has five.

Fetching the instructions that will soon be executed requires knowing the address of those instructions. In the case of function calls the destination address is usually encoded as part of the instruction; however, the function return address is usually held on the stack (along with other housekeeping information). Maintaining a second stack, containing only function return addresses, has been proposed, along with speculative execution (and stack repair if the speculation does not occur along the control flow path finally chosen[1251]).

Calder, Grunwald, and Srivastava[191] studied the behavior of branches in library functions, looking for common patterns that occurred across all calls.

Coding Guidelines

The **`continue`** and **`break`** statements are a form of **`goto`** statement. Some developers consider them to be a *structured goto* and treat them differently than a **`goto`** statement. The controversy over the use of the **`goto`** statement has not abated since Dijkstra's, now legendary, letter to the editor was published in 1968.[357] Many reasons have been given for why source code should not contain **`goto`** statements; Dijkstra's was based on human cognition. Knuth argued that in some cases use of **`goto`** provided the best solution.[751]

Edsger W. Dijkstra

> *My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

The heated debate on the use of the **`goto`** statement has generated remarkably little empirical research.[479] Are guideline recommendations against the use of **`goto`** simply a hang over from the days when developers had few structured programming constructs (e.g., compound statements) available in the language they used, or is there a worthwhile cost/benefit in recommending against their use?

It is possible to transform any C program containing jump statements to one that does not contain any jump statements. This may involve the introduction of additional **`while`** statements, **`if`** statements, and the definition of new objects having a boolean type. An algorithm for performing this transformation, while maintaining the topology of the original flow graph and the same order of efficiency, is given by Ashcoft and Manna.[61] Ammaraguellat[28] gives an algorithm that avoids code replication and normalizes all control-flow cycles into single-entry single-exit **`while`** loops. In practice automated tools tend to take a simpler approach to transformation.[397] The key consideration does not appear to be the jump statement itself, but the destination statement relative to the statement performing the jump. This issue is discussed elsewhere.

jump 1783 statement causes jump to

Usage

Numbers such as those given in Table 1782.1 and Table 1782.2 depend on the optimizations performed by an implementation. For instance, unrolling a frequently executed loop will reduce the percentage of branch instructions.

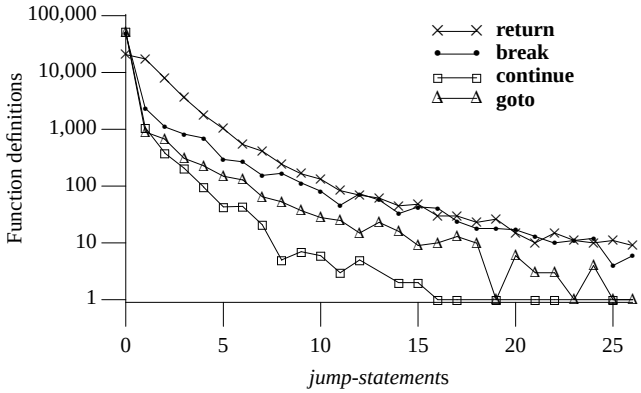


Figure 1782.1: Number of function definitions containing a given number of *jump-statement*s. Based on the translated form of this book's benchmark programs.

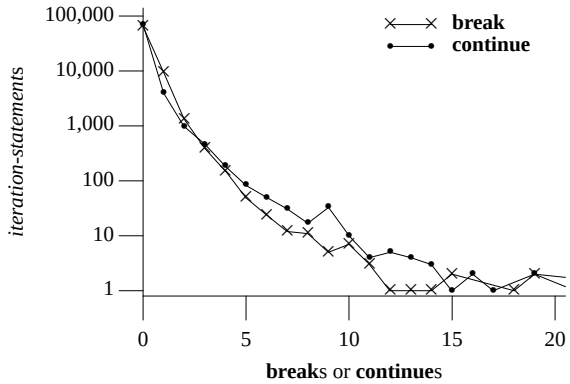


Figure 1782.2: Number of *iteration-statement* containing the given number of **break** and **continue** Based on the visible form of the `.c` files.

Table 1782.1: Dynamic occurrence of different kinds of instructions that can change the flow of control. *%Instructions Altering Control Flow* is expressed as a percentage of all executed instructions. All but the last row are expressed as percentages of these, control flow altering, instructions only. The kinds of instructions that change control flow are: conditional branches *CB*, unconditional branches *UB*, indirect procedure calls *IC*, procedure calls *PC*, procedure returns *Ret*, and other breaks *Oth* (e.g., signals and **switch** statements). *Instructions between branches* is the mean number of instructions between conditional branches. Based on Calder, Grunwald, and Zorn.[192]

Program	%Instructions Altering Control Flow	%CB	%UB	%IC	%PC	%Ret	%Oth	%Conditional Branch Taken	Instructions Between Branches
burg	17.1	74.1	6.9	0.0	9.5	9.5	0.0	68.8	7.9
ditroff	17.5	76.3	4.2	0.1	9.7	9.8	0.0	58.1	7.5
tex	10.0	75.9	10.7	0.0	5.8	5.8	1.9	57.5	13.2
xfig	17.5	73.6	7.7	0.6	8.6	9.2	0.3	54.8	7.8
xtex	14.1	78.2	8.5	0.2	6.0	6.2	1.0	53.3	9.1
compress	13.9	88.5	7.6	0.0	2.0	2.0	0.0	68.3	8.1
eqntott	11.5	93.5	2.1	1.5	0.7	2.2	0.0	90.3	9.3
espresso	17.1	93.2	1.9	0.1	2.3	2.4	0.1	61.9	6.3
gcc	16.0	78.9	7.4	0.4	6.1	6.5	0.8	59.4	7.9
li	17.7	63.9	8.7	0.4	12.9	13.2	0.9	49.3	8.9
sc	22.3	83.5	3.9	0.0	6.3	6.3	0.0	64.3	5.4
Mean	15.9	80.0	6.3	0.3	6.3	6.6	0.5	62.4	8.3

Table 1782.2: Number of static conditional branches sites that are responsible for the given quantile percentage of dynamically executed conditional branches. For instance, 19 conditional branch sites are responsible for over 50% of the dynamically executed branches executed by burg. *Static count* is the total number of conditional branch instructions in the program image. Of the 17,565 static branch sites, 69 branches account for the execution of 50% of all dynamic conditional branches. Not all branches will be executed during each program execution because many branches are only encountered during error conditions, or may reside in unreachable or unused code. Based on Calder, Grunwald, and Zorn.[192]

Program	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	99%	100%	Static count
burg	1	3	5	9	19	33	58	95	135	162	268	859	1,766
ditroff	3	11	19	28	38	50	64	91	132	201	359	867	1,974
tex	3	7	15	26	39	58	89	139	259	416	788	2,369	6,050
xfig	8	31	74	138	230	356	538	814	1,441	2,060	3,352	7,476	25,224
xtex	2	8	15	22	36	63	104	225	644	1,187	2,647	6,325	21,597
compress	1	2	2	3	4	5	6	8	12	14	16	230	1,124
eqntott	1	1	1	2	2	2	2	3	14	42	72	466	1,536
espresso	4	10	19	30	44	63	88	121	163	221	470	1,737	4,568
gcc	13	38	77	143	245	405	641	991	1,612	2,309	3,724	7,639	16,294
li	2	4	7	11	16	22	29	38	52	80	128	557	2,428
sc	2	3	4	6	9	16	30	47	76	135	353	1,465	4,478
Mean	3	10	21	38	62	97	149	233	412	620	1,107	2,726	7,912

Semantics

jump statement causes jump to

A jump statement causes an unconditional jump to another place. 1783

Commentary

For the **goto**, **continue**, and **break** statements the other place is within the function body that contains the statement.

Other Languages

Some languages allow labels to be treated as types. In such languages jump statements can jump between functions (there is usually a requirement that the function jumped to must have an active invocation in the current call chain at the time the jump statement is executed). In Algol 68 a label is in effect a function that jumps to that label. It is possible to call that function or take its address. In C terms:

```
somewhere:
    /* ... */
    if (problem)
       somewhere();    /* Same as "go to somewhere". */
    /* ... */
    void (*fp)(void) = &somewhere;
```

Common Implementations

The **goto**, **continue** and **break** statements usually map to a single machine instruction, an unconditional jump. Jumping out of a compound statement nested within another compound statement often creates a series of jumps. For instance, in the following example:

```
if (x)
   {
   for (;;)
      {
      some_code;
      if (y)
         break; /* Jump out of loop. */
      }
/* Place A */
   }
```

```
11  else
12     some_more_code;
13  /* Place B */
```

the branch instruction out of the loop, generated by the **break** statement, is likely to branch to an instruction (at place A) that branches over the **else** arm of the **if** statement (to place B). One of the optimization performed by many translators is to follow a *jump chain* to find the final destination. The destination of the branch instruction at the start of the chain being modified to refer to this place.

Many processors have span-dependent branch instructions. That is, a short-form (measured in number of bytes) that can only branch relatively small distances, while a long-form can branch over longer distances. When storage usage needs to be minimized it may be possible to use a jump chain to branch to a place using a short-form instruction, rather than a direct jump to that place using a long-form instruction (at the cost of reduced execution performance).[846]

Coding Guidelines

The term *spaghetti code* is commonly used to describe code containing a number of jump statements whose various destinations cause the control flow to cross the other control flows (a graphical representation of the control flow, using lines to represent flow, resembled cooked spaghetti, i.e., it is intertwined).

Jumps can be split into those cases where the destination is the most important consideration and those where the jump/destination pair need to be considered— as follows:

- *Jumping to the start/end of a function/block*— the destination being in the same or outer block relative to the jump . This has a straight-forward interpretation as restarting/finishing the execution of a function/block. The statement jumped to may not be the last (some termination code may need to be executed, or a guideline recommendation that functions have a single point of exit may cause the label to be on a **return** statement).
- *Jumping into a nested block.* This kind of jump/destination pair is the one most often recommended against.
- *Jumping out of a nested block.* This kind of jump/destination pair may be driven by the high cost of using an alternative construct. For instance, adding additional flags to cause a loop to terminate may not introduce excessive complexity when a single nesting level is involved.
- *Jumping within the same block.* This is the most common kind of **goto** statement found in C source (see right plot of Figure 1783.2).

1790 goto EXAMPLE

1766 iteration statement executed repeatedly

Jump statements written by the developer can create a flow of control that is that requires a lot of effort to comprehend. Some guideline documents recommend against the use of any jump statement (including a **return**, unless it is the last statement within a function). Comprehending the flow of control is an important part of comprehending a function. The use of jump statements can increase the cost of comprehension (by increasing the complexity of the flow of control) and can increase the probability that the comprehension process reaches the wrong conclusion (unlike other constructs that change the flow of control, there are not usually any addition visual clues, e.g., indentation, in the source signaling the presence of a jump statement). However, there is no evidence to suggest that the cost of the alternatives is any lower and consequently no guideline recommendation is made here.

Any statements that appear after a *jump-statement*, in the same block, are dead code.

190 dead code

Usage

A study by on Gellerich, Kosiol, and Ploedereder[479] analyzed **goto** usage in Ada and C. In the translated form of this book's benchmark programs 20.6% of **goto** statements jumped to a label that occurred textually before them in the source code.

1784 134) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop;

footnote 134

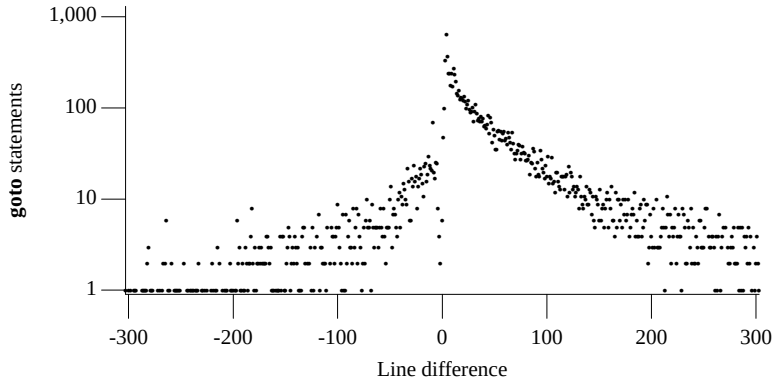


Figure 1783.1: Number of `goto` statements having a given number of visible source lines between a `goto` statement and its destination label (negative values denote backward jumps). Based on the translated form of this book's benchmark programs.

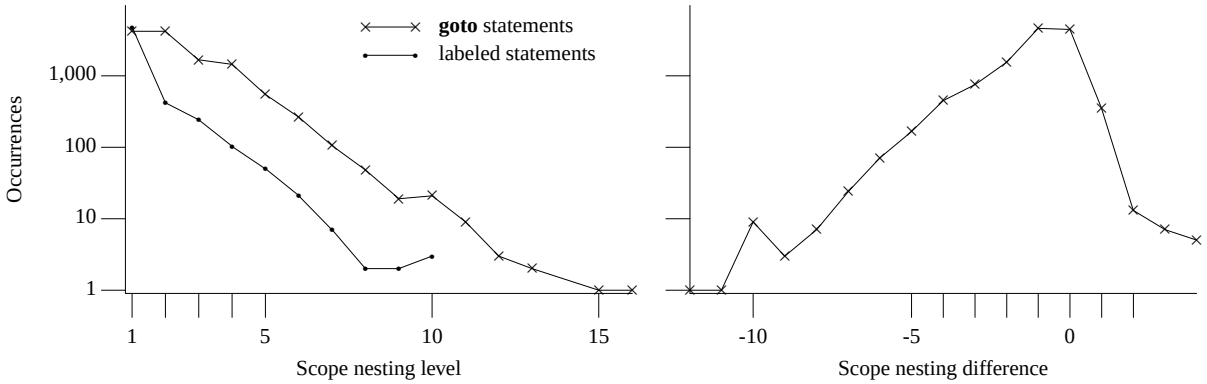


Figure 1783.2: Number of `goto` statements and labels having a given scope nesting level (nesting level 1 is the outermost block of a function definition), and on the right the difference in scope levels between a `goto` statement and its corresponding labeled statement (negative values denote a jump to a less nested scope). Based on the translated form of this book's benchmark programs.

Commentary

Technically the evaluation *clause-1* need have no connection with the contents of the body of the loop. This wording expresses the view of C committee about how they saw this construct being used by developers.

C90

Support for declaring variables in this context is new in C99.

C++

The C++ Standard does not make this observation.

---

the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; 1785

Commentary

In C90 this wording appeared in a footnote, where it was an observation about the equivalence between a **for** statement and a **while** statement. In C99 this wording plays the role of a specification.

controlling expression 1775 for statement

Coding Guidelines

The discussion on the controlling expression in an **if** statement is applicable here.

---

and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration. 1786

Commentary

The *incrementing* operation referred to here is the concept of a loop control variable having its value increased (often by one). Decrementing (i.e., decreasing the value of the loop control variable) is a less commonly specified operation (as is walking a linked list). There is often a causal connection between the operand that is incremented and one of the operand appearing in *expression-2*.

6.8.6.1 The goto statement

Constraints

1787 The identifier in a **goto** statement shall name a label located somewhere in the enclosing function.

goto statement

Commentary

The situation where a label having a corresponding name does not occur within the enclosing function is likely to be some kind of fault. Having translators issue a diagnostic is probably the most useful behavior for the standard to specify.

Other Languages

Some languages that support nested function definitions (e.g., Pascal) only require that the label name be visible (i.e., it is possible to jump to a label in a different function). Other languages (e.g., Algol 68) give labels block scope, which restricts its visibility. Perl has a form of **goto** that causes the function named by the label to be called. However, when that function returns control is returned to the function that called the function that performed the **goto** (i.e., the behavior is as-if control returns to the **goto** which then immediately performs a return).

1788 A **goto** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

goto past variably modified type

Commentary

Storage allocation for objects having a variably modified type differs from storage allocation for objects having other types in that it requires internal implementation housekeeping code to be executed, when its definition is encountered during program execution, for it to occur (storage for objects having other types can be reserved at translation time). Similarly, storage deallocation requires housekeeping code to be executed when the lifetime of an object having a variably modified type ends (implementation techniques that have no dependency between execution of an objects definition and the termination of its lifetime discussed elsewhere); storage deallocation for objects having other types does not require any execution time actions.

464 VLA lifetime starts/ends
1354 storage layout

464 VLA lifetime starts/ends
1749 switch past variably modified type

This constraint, along with an equivalent one for the **switch** statement, provides a guarantee to implementations that if the storage deallocation code is executed, then the storage allocation code will have been previous executed (i.e., there is no way to bypass it in a conforming program). Jumping from outside of the scope of an identifier denoting such an object to inside the scope of that identifier bypasses execution of its definition. Bypassing the definition might be thought to be harmless if the object is never referenced. However, storage for the object has to be deallocated when its lifetime terminates, which is a (implicit) reference to the object.

This constraint does not require the identifier to be visible at the jumped to label, only that the label be within the scope of the identifier. Also, there is no constraint prohibiting the block containing the **goto** statement from containing an object defined to have a variably modified type and the destination of the jump to be outside of that block.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard. However, the C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that

being visible rather than *in scope* more accurately reflects the intent):

6.7p3 *A program that jumps[77] from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).*

A C function that performs a jump into a block that declares an object with an initializer will cause a C++ translator to issue a diagnostic.

```
1  void f(void)
2  {
3  goto lab;    /* strictly conforming */
4               // ill-formed
5  int loc = 1;
6
7  lab: ;
8  }
```

Semantics

goto causes unconditional jump

A **goto** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function. 1789

Commentary

dead code 190

Unless prefixed by a label, statements appearing after a **goto** statement, in the same block, are dead code.

Other Languages

Some languages (e.g., Pascal) support jumping to statements contained within other functions.

Common Implementations

There are a number of practical algorithms for analyzing program control flow that depend on the graph of the control flow being reducible[17,558] (i.e., not irreducible; an irreducible graph might be thought of one that contains a cycle with multiple entry points). To create an irreducible flow graph in C requires the use of the **goto** statement, for instance:

```
1        if (a > b)
2           goto L_1;
3  L_2: a++;
4  L_1: if (a < 3)
5           goto L_2;
```

Although algorithms are available for transforming an irreducible flow graph into a reducible one,[1388] many optimizers and static analyzers don't perform such transformations (because of the potentially significant increase in code size created by the node splitting algorithms,[205] and because it is believed that function definitions rarely have a control flow that is irreducible[14]). For this reason the quality of the machine code generated for the few functions that do contain an irreducible control flow graph can be much lower than that for other functions (because irreducibility can prevent some of the information needed for optimization being deduced).

goto EXAMPLE

EXAMPLE 1 It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions: 1790

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.

3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **continue** statements, for example.)

```
/* ... */
goto first_time;
for (;;) {
        // determine next operation
        /* ... */
        if (need to reinitialize) {
                // reinitialize-only code
                /* ... */
        first_time:
                // general initialization code
                /* ... */
                continue;
        }
        // handle other operations
        /* ... */
}
```

Commentary

While this might seem like a contrived example, the same could probably be said for all examples of the use of the **goto** statement.

1791 EXAMPLE 2

EXAMPLE goto variable length

A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
goto lab3;                      // invalid: going INTO scope of VLA.
{
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4;              // valid: going WITHIN scope of VLA.
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
}
goto lab4;                      // invalid: going INTO scope of VLA.
```

Commentary

It is a constraint violation for a **goto** statement to jump past any declarations of objects with variably modified types.

1788 goto past variably modified type

6.8.6.2 The continue statement

Constraints

1792 A **continue** statement shall appear only in or as a loop body.

continue shall only appear

Commentary

The behavior of the **continue** statement is only defined in this context.

Coding Guidelines

Some coding guideline documents recommend against the use of the **continue** statement. The rationale appears to be based on an association being made with the **goto** statement.

The **continue** statement can be thought about either in implementation terms (i.e., jump to just before the end of the iteration statement that contains it) or conceptual terms (i.e., execute the next iteration of the loop).

Neither of these are likely to require significant effort to comprehend. While it may not require a significant amount of effort to comprehend, readers still have to notice its existence in the source code. The **continue** statement has the same reader visibility issues as all jump statements.

jump 1783 statement causes jump to

Semantics

A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; 1793

Commentary

The **continue** statement performs the dual role of **goto** statement and virtual label creator.

Rationale The C89 Committee rejected proposed enhancements to **continue** and **break** which would allow specification of an iteration statement other than the immediately enclosing one on grounds of insufficient prior art.

---

that is, to the end of the loop body. 1794

Commentary

The following C statement clarifies what is meant by "end of the loop body".

More precisely, in each of the statements 1795

```
while (/* ... */) {    do {                    for (/* ... */) {
   /* ... */                 /* ... */              /* ... */
   continue;                 continue;              continue;
   /* ... */                 /* ... */              /* ... */
contin: ;                contin: ;               contin: ;
}                       } while (/* ... */);     }
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin;**.[135)]

Commentary

These equivalent mappings describe an effect, not an implementation technique. The **for** statement example could be rewritten as:

```
1  {
2  clause_1;
3  while (expression_2)
4     {
5       {
6       /* ... */
7       continue;
8       /* ... */
9       contin: ;
10      }
11    expression_3;
12    }
13 }
```

C++

The C++ Standard uses the example (6.6.2p1):

6.6.2p1

```
while (foo) {          do {           for (;;) {
  {                     {              {
   // ...                // ...         // ...
  }                     }              }
contin: ;              contin: ;      contin: ;
}                      } while (foo);  }
```

The additional brace-pair are needed to ensure that any necessary destructors (a construct not supported by C) are invoked.

Coding Guidelines

Many developers have a mental model in which the **continue** statement jumps to the top of the loop. This model makes sense in that there is usually no information at the end of the loop body that developers need to consider (use of the **do** statement is relatively rare).

6.8.6.3 The break statement

Constraints

1796 A **break** statement shall appear only in or as a switch body or loop body.

Commentary

The behavior of the **break** statement is only defined in these contexts.

Other Languages

In most languages the arms of a **switch** body are syntactic units, each having an implicit **break** statement after the last statement. There is no requirement for a construct having the behavior of the C **break** statement, in a **switch** body context.

1739 selection statement syntax

Coding Guidelines

The coding guideline discussion for the **continue** statement is also applicable to the **break** statement. In the context of a **switch** statement developers are invariably going to need to use to **break** statements. Given that many of the reader comprehension issues apply to all uses of jump statements, any argument against the use of **break** statements could also be used to argue that **switch** statements should not be used.

1792 continue shall only appear

Semantics

1797 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

Commentary

Using the analogy given earlier for the **continue** statement:

```
1  {                       {                       {
2  while (/* ... */) {     do {                    for (/* ... */) {
3    /* ... */               /* ... */               /* ... */
4    break;                  break;                  break;
5    /* ... */               /* ... */               /* ... */
6    }                       } while (/* ... */);    }
7  brea: ;                 brea: ;                 brea: ;
8  }                       }                       }
```

The outer pairs of braces not appearing in the source code, they have been added here to show an affect.

Other Languages

Some languages allow blocks to be labeled. This label name can then appear in a statement (sometimes using the keyword **exit**), indicating that execution of that block is to terminate.

In most other languages **case** and **default** labels are part of the syntax of the **switch** statement. When the statement associated with these labels completes execution flow of control is defined to continue after the **switch** statement (there is an implicit break statement). These languages do not usually support a special statement whose purpose is to terminate execution of a **switch** statement (a **goto** statement could be used to do this). In BCPL the **ENDCASE** statement is equivalent to the C **break** statement within a **switch** statement (the BCPL **break** statement terminates execution of loops only).

---

footnote 135

135) Following the **contin:** label is a null statement. 1798

Commentary

That is, there are no statements from the loop body that were written by the developer.

6.8.6.4 The return statement

Constraints

---

return void type

A **return** statement with an expression shall not appear in a function whose return type is **void**. 1799

Commentary

base document 1

The base document did not define any semantics for this case (and simplifying automatic C source code generators was not considered to be sufficiently important for the committee to define any).

C++

6.6.3p3 *A return statement with an expression of type "cv **void**" can be used only in functions with a return type of cv void; the expression is evaluated just before the function returns to its caller.*

Source developed using a C++ translator may contain **return** statements with an expression returning a void type, which will cause a constraint violation if processed by a C translator.

```
1  void f(void)
2  {
3  return (void)4; /* constraint violation */
4                  // does not change the conformance status of a program
5  }
```

Other Languages

Many other languages support constructs called *procedure*s or *subroutine*s to take the role performed by function returning type **void**. The **return** statements appearing within these constructs are not allowed to include an expression.

---

return without expression

A **return** statement without an expression shall only appear in a function whose return type is **void**. 1800

Commentary

base document 1

The base document did not provide a mechanism enabling function declarations to explicitly specify that they did not return a value. It was common practice to omit the return type in the function definition to indicate that the function did not return a value. Unfortunately it was also common practice for developers to omit a return type and rely on the translator supplying an implicit **int** type. The type **void** was introduced into C by the C90 Standard.

type specifiers 1382 possible sets of

This constraint is new in C99 and ties in with the removal of support for *implicit int* in declarations.

function termination 1844 reaching }

The behavior that occurs after executing the last statement in a function definition, when it is not a **return** statement, is discussed elsewhere.

C90

This constraint is new in C99.

```
1  int f(void)
2  {
3  return; /* Not a constraint violation in C90. */
4  }
```

Other Languages

Many languages do not permit the expression in a **return** statement to be omitted when the containing function is defined to return a value.

Common Implementations

Some C90 implementations provided an option to switch on the diagnosing of additional constructs. The requirement specified in this constraint was often one of these additional constructs.

Coding Guidelines

This constraint is new in C99 and the majority of existing, C90, implementations do not issue a diagnostic for a violation of it. Any coding guideline documents that continue to be based on the C90 Standard (e.g., MISRA) might like to consider including a guideline recommendation along the lines of this constraint. The practice of not specifying an expression when it is known that the caller does not make use of the returned value also occurs in existing code, this issue is discussed elsewhere.

0 MISRA

1844 function termination reaching }

Usage

The translated form of this book's benchmark programs contained 19 instances of a **return** statement without an expression appearing in a function whose return type was **void**.

Semantics

1801 A **return** statement terminates execution of the current function and returns control to its caller.

Commentary

The **return** statement is one of the two constructs that can be used to terminate execution of the current function (use of `longjmp` is relatively rare).

Other Languages

Some languages (e.g., Pascal) support nested functions and what are known as *non-local goto* statements. Within a body of a nested function it is possible to perform non-local goto to a label visible in the body of an outer function. Fortran supports what is known as an *alternative return*. This kind of return (it is only supported for subroutines, not functions) causes control to resume at some location other than the statement following the one that performed the call. The alternative locations at which execution resume are passed as arguments to the call.

Common Implementations

Many processors include a return instruction that is the inverse of the one used to perform a call. A return instruction has to reinstate the stack (the most common form used to implement function call and return) to the state it was in prior to the call. This is often simply a matter of resetting the stack pointer to its value prior to the call. The return address may be loaded into a register (and an indirect jump performed) or popped from the stack (many processors contain an instruction that pops an address off the stack and jumps to it).

Depending on the calling conventions used either the caller or callee will restore any registers whose contents were saved prior to the call. Implementations where the callee restores the registers are likely to translate a **return** statement into a branch to the end of the function,[325] reducing duplication of the common housekeeping code.

1004 register function call housekeeping

It is unlikely that storage allocated to objects having variably modified types will be treated any differently than storage allocated to objects having other types (because, like them, it is allocated on the stack).

A function may have any number of **return** statements. 1802

Commentary

When the C language was first specified it was not uncommon for languages definitions to specify that a **return** statement ended the body of a function/procedure/subroutine.

C++

The C++ Standard does not explicitly specify this permission.

Other Languages

Some languages (e.g., Pascal) have no **return** statement. A function (or procedure/subroutine) returns when the flow of control reaches the end of its body.

Coding Guidelines

Some coding guideline documents specify that function definitions should contain a single **return** statement. Adhering to such a recommendation may involve using either a **goto** statement or additional flags (that prevent statements following the return decision point being executed). The cost of using these alternative constructs does not appear to be less than that associated with using multiple **return** statements.

iteration statement 1766 executed repeatedly

If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. 1803

Commentary

C specifies that functions return values, not references. Implementations need to act as if temporary storage was used to hold the value being returned by a function. In an assignment statement this temporary object and the object being assigned to do not overlap.

footnote 1806 136

C++

The C++ Standard supports the use of expressions that do not return a value to the caller.

return void type 1799

Other Languages

In some languages (e.g., Fortran and Pascal) the last value assigned to an identifier, whose spelling is the same as that of the called function, is the result of a function call. Pascal does not support a **return** statement and execution of a function does not terminate until control reaches the end of that function. In Algol 68 the value returned by a function is the value of the last expression in its body (rather like the behavior of the `gcc` compound expression).

compound expression 1313

Common Implementations

Values having scalar type are usually returned in a register. Those having a structure type are usually returned in an area of storage allocated by the caller (often passed as a hidden parameter to the called function).

---

return implicit cast

If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[136)] 1804

Commentary

This situation can only occur if the expression has a scalar type. As the response to DR #094 pointed out, this *as if* assignment implies that the constraints given for the assignment operator also apply here.

assignment operator modifiable lvalue 1289

C++

In the case of functions having a return type of *cv* **void** (6.6.3p3) the expression is not implicitly converted to that type. An explicit conversion is required.

Coding Guidelines

The guideline recommendations applicable here are the same as those that apply to any implicit conversions.

operand convert automatically 653

1805 EXAMPLE In:

```
struct s { double i; } f(void);
union {
        struct {
                int f1;
                struct s f2;
        } u1;
        struct {
                struct s f3;
                int f4;
        } u2;
} g;

struct s f(void)
{
        return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

Commentary

Neither is there any undefined behavior if the function `f` is defined as an inline function. 1526 inline function

1806 136) The **return** statement is not an assignment. footnote 136

Commentary

The difference between the **return** statement and the assignment operator is that in the former case the value is required to act as if it were held in temporary storage before it is assigned (rather like the difference between the `memcpy` and `memmove` library functions). In an assignment statement it is intended that implementations be able to perform the operation without the need of temporary storage. Possible differences in behavior can only occur when operating storage for the two operands overlaps.

C90

This footnote did not appear in the C90 Standard. It was added by the response to DR #001.

C++

This distinction also occurs in C++, but as a special case of a much larger issue involving the creation of temporary objects (for constructs not available in C).

A return statement can involve the construction and copy of a temporary object (12.2). 6.6.3p2

Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), . . . 12.2p1

Common Implementations

Translators that inline function calls, returning a structure or union type, have to be careful about how they handle **return** statements. Mapping them to an assignment may not produce the same affect as a function call. 1529 inline suggests fast calls

The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return. 1807

Commentary

assignment 1304 value overlaps object

This is effectively a requirement on implementations to ensure that the behavior is well defined (not undefined as specified elsewhere).

The representation of floating-point values may have wider range or precision and is determined by FLT_EVAL_METHOD. 1808

Commentary

floating 353 operands evaluation format

This sentence calls out behavior that is specified elsewhere as applying to the expression in a **return** statement.

This sentence was added by the response to DR #290.

A cast may be used to remove this extra range and precision. 1809

Commentary

return 1804 implicit cast

An implicit conversion is only performed on the value in a **return** statement if the type of this value is different from that of the return type.

This sentence was added by the response to DR #290.

6.9 External definitions

translation unit syntax
external declaration syntax

1810

```
translation-unit:
                external-declaration
                translation-unit external-declaration
external-declaration:
                function-definition
                declaration
```

Commentary

syntactically 138 analyzed preprocessor 1854 directives syntax
translation unit 110 known as

A *translation-unit* is the terminal production of C syntax. By the time a translator needs to perform syntax analysis the preprocessor directives, such as **#include** and **#define**, have been deleted (C syntax essentially has two terminal productions, the other being *preprocessing-file*). The term *translation unit* applies to the result of preprocessing a source file. This syntax requires that a translation unit contain at least one declaration (i.e., a source file that only contains preprocessing directives is not conforming). One of the reasons for this requirement is that some translators require the object file they create to contain at least one symbol.

The mechanism by which identifiers declared in separate translation units are made to refer to each other is linkage.

linkage 420

C++

The C++ syntax includes function definitions as part of declarations (3.5p1):

3.5p1 *translation-unit:* *declaration-seq*$_{opt}$

While the C++ Standard differs from C in supporting an empty translation unit, this is not considered a significant difference.

Other Languages

Some languages (e.g., Cobol and the first Pascal Standard) do not support any external definitions because they require all of a programs source to be translated at the same time. While other languages (e.g., Ada)

provide sophisticated separate compilation mechanisms. Some languages impose a structure, or ordering, on the components in a translation unit. For instance, Pascal requires constants to be declared first, followed by type declarations, then variable declarations and then function/procedure declarations.

Coding Guidelines

Although the interface issues are generally considered to be a significant source of faults in software there have been few empirical studies of these kinds of fault. Although a few studies have looked at source code level (e.g., incorrect number of parameters),[1344] most have tended to investigate higher level algorithmic or design issues.[1077]

Prototypes

prototypes cost/benefit

Use of function prototypes are strong recommended, if not mandated, by nearly every coding guideline document. Surprisingly there is little empirical evidence to support the claimed benefits over not using a prototype (because nobody has done the experiments). The one study that has been performed,[1121] using experienced developers, showed that use of prototypes did increase productivity. The experiment was not typical of industry practice in that subjects were provided with a paper listing of the names of all types, constants, and functions that might be required. If a program is being written from scratch the cost of using prototypes is minimal and is probably recouped the first time a translator diagnoses a mismatch between the number of parameters in a function definitions and the number arguments given in a call to it. Use of prototypes are the only economically worthwhile option.

Cg 1810.1

An *external-declaration* that declares a function shall always contain a parameter type list (i.e., a function prototype).

However, the cost/benefit decision is not so clear-cut when modifying existing code that contains old-style definitions. When making a small modification to existing code, for instance adding a function call, the impact of changing the called function (to use a prototype in its definition) may extend throughout a programs source tree. Other source files, that are not directly touched by the original modification, may contain calls to the function whose definition has been changed (to use a prototype).

Should function definitions in existing code, that don't use prototypes, be changed to use prototypes on an individual basis, as small modifications are made to the source? Or should such changes to function definitions only be performed as part of a more global reengineering of the source code? The answer to these questions will depend on what phase of its life cycle a program is in, the current commercial environment, and technical factors such as the product development environment and the product testing procedures used.

Ordering of declarations

Within a source file external declarations usually occur in some kind of recognizable order. For instance, function definitions usually appear after all of the other kinds of declarations and definitions. There is a benefit in using a recognizable order for declarations and definitions, it is information that readers can use when searching for the declaration of a particular identifier. There are a number of factors that might be considered when ordering identifier declarations, including:

- Ordering by C language attributes associated with the declared identifier. For instance, one ordering might be: some preprocessor directives (see elsewhere for a discussion on the ordering of preprocessor directives), followed by typedef names, definitions of objects with external linkage, then objects with internal linkage, and so on.

1854 preprocessor directives syntax

- Ordering by algorithmic or implementation attributes associated with the declared identifier. For instance, the macro names, typedef names, and objects color, in a translation unit that handles printing, might all be declared together in one section of the source file.

Both ordering have their own costs and benefits. For instance, ordering by C language attribute allows identifiers having that attribute to be quickly located within a source file. However, reading the declaration

of an identifier may create the need to obtain information about identifiers associated with it (e.g., type declarations). An ordering that places declarations of identifiers sharing some form of algorithmic or implementation attributes together could reduce effort for searches based on these attributes.

Selecting an optimal declaration ordering requires information on the search patterns of future readers of the source. Given that this information is unlikely to be available, the most that the authors of declarations can do is ensure they use an ordering that future readers will recognize and be able to use when searching for identifier declarations.

declarations in which source file

Which declarations in which source file?

Technically any external declaration of an object or function could be defined in any of the sources files used to build a program (provided the necessary type definitions were visible). However, experience shows that when writing a new declaration, developers often attempt to put it in a source file containing other external declarations that are considered to be related to it. The set of external object and function contained in a translation unit are often considered to be members of a category. For instance, the contents of the library headers defined by the C Standard (e.g., string handling, time information, character handling, etc.).

categorization developers organized knowledge

This developer behavior of organizing object and function definitions into categories is a special case of general human categorization behavior. People use information about category membership in a number of ways. For instance, as an aid to recall, or to deduce properties or behavior of a category member based on their knowledge of other category members.

If the classification process is driven by individual choice, then it is necessary to ask to what extent the classification chosen by one developer is of benefit to another developer. At the time of this writing there have been no studies of developer classification behavior that might be used to suggest an answer to this question (although the semantic processed involved in creating identifier spellings may be related).

A number of mathematical methods for *software clustering* (as the field has become known) have been proposed. These methods are generally legacy systems oriented in that they take the source of an existing program and attempt to find the subsystems from which it is composed.[1462] They vary in the algorithms used and the measure of source code similarity used. Mitchell and Mancoridis[944] empirically compare various algorithms and similarity measures. Fasulo[410] provides an analysis of recent mathematical work on general clustering algorithms (i.e., not software specific).

application evolution

A common characteristic of programs that have an active community of users is that they continue to grow and evolve. For instance, an analysis, by Godfrey and Tu[499] of 96 releases of the source of the Linux kernel found that the number of uncommented lines of code closely fitted the equation $0.21 \times X^2 + 252 \times X + 90055$, where X is the number of days since release 1.0. At the other end of the scale are programs that are rarely used and whose source is relatively unchanging. The source files are changed tend to be those dealing with areas that are important to business.[720]

When adding new functionality to existing code developers are faced with making a cost/benefit analysis. Should they perform any restructuring that they feel is necessary (e.g., breaking up the contents of large source file into smaller source files, each representing some facet of the category represented by the original source), an investment cost that may not be repaid by a future benefit, or should they ignore future costs and minimize current costs (i.e., no restructuring)?

These issues are very complex and at the time of this writing it does not appear to be possible to give any simple guideline recommendation on which source file object and function declarations should appear in.

coupling

The coupling between two source files (i.e., the extent to which their objects and functions each other) has been found to affect the number of faults in a program.[97, 1212] The commercial benefits, to hardware vendors, of minimizing the coupling between separate units is well documented.[87] However, the benefits may not unconditionally apply to software[162] and minimizing coupling may even increase costs in some cases.[90] Until the costs and benefits associated with coupling are better understood it is not possible to know if any guideline recommendation would be worthwhile.

By default, in C, external declarations of identifiers for objects and functions are visible outside of the translation unit that declares them (i.e., they have external linkage). Explicitly declaring identifiers to have

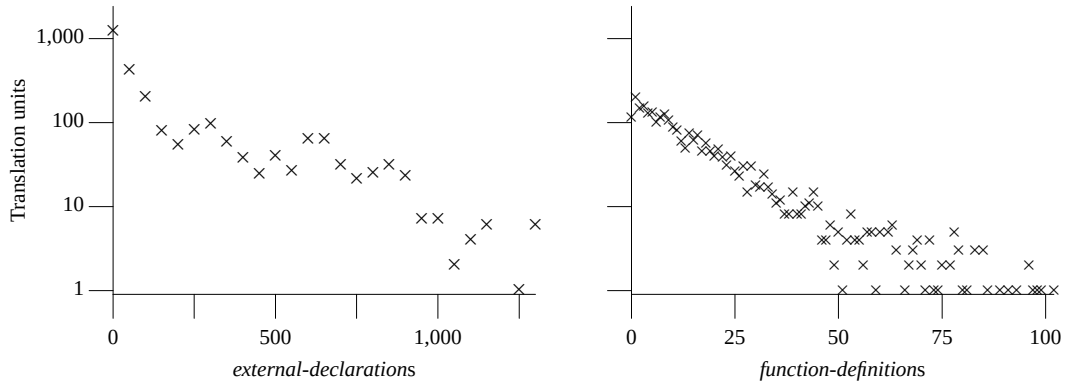


Figure 1810.1: Number of translation units containing a given number of *external-declaration*s and *function-definition*s declarations (rounded to the nearest fifty and excluding identifiers declared in any system headers that are **#include**d). Based on the translated form of this book's benchmark programs.

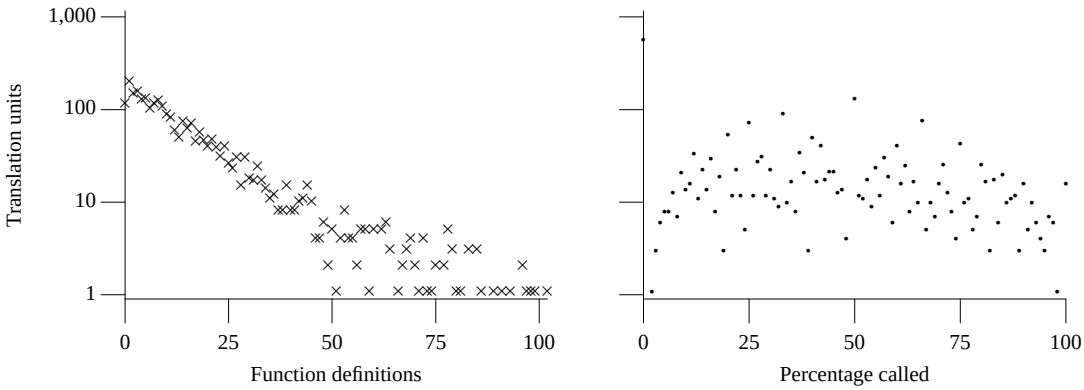


Figure 1810.2: Number of translation units containing a given number of function definitions and percentage of functions that are called within the translation unit that defines them. Based on the translated form of this book's benchmark programs.

internal linkage has the benefit of reducing the probability of name clashes with identifiers declared in other translation units. However, while the developer cost of typing the keyword **static** is negligible, the cost of working out which identifiers can be so declared may be nontrivial. Since the cost of a guideline recommending the use of internal linkage, where possible, may often be greater than the benefit, no such recommendation is made here.

Usage

On a large development project it is possible that more than one person will write some set of functions performing similar operations. This duplication of functionality occurs at a higher-level than copying and reusing sequences of statements (discussed elsewhere), it is a concept that is being duplicated. Marcus and Maletic[895] used latent semantic analysis to identify related source files (what they called *concept clones*). Source code identifiers and words in comments were used as input to the indexing process. An analysis of the Mozilla source code highlighted two different implementations of linked list functions and four files that contained their own implementations.

1821 duplicate code
792 latent semantic analysis

Constraints

1811 The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.

external declaration not auto/register

Commentary

While it would have been possible to allow the storage-class specifier **register** to appear on some external declarations (e.g., those having internal linkage) the Committee did not.

C++

The C++ Standard specifies where these storage-class specifiers can be applied, not where they cannot:

7.1.1p2 *The **auto** or **register** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).*

A C++ translator is not required to issue a diagnostic if the storage-class specifiers **auto** and **register** appear in other scopes.

Common Implementations

Some implementations[1437] take the complete program into account, during register allocation, rather than just one individual function definitions at a time. However, this is one case where developers may still be able to use their knowledge of program behavior to make better use of register resources (high-quality automatic register allocation is very dependent on the use of program execution traces). `gcc` supports the **register** storage class appearing in the declaration of objects at file scope. However, the register to be used needs to be explicitly specified, for instance in:

```
1  register int *ipc asm("a5");
```

the register `a5` is dedicated to holding the value of the object `ipc`.

---

definition one external

There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. 1812

Commentary

external definition 1817, tentative definition 1849, extern 426, identifier linkage same as prior declaration

This C sentence is referring to an *external definition*, as defined below. It is possible for there to be no explicit external definition, in which case a tentative one is implicitly created. It is possible for the same *external definition* to be declared more than once in the same translation unit.

Other Languages

linkage 420

Most languages only allow one definition of any kind of identifier. Fortran allows more than one and linkers are expected to pick a unique instance.

---

internal linkage exactly one external definition

Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit. 1813

Commentary

external definition 1817

This C sentence is referring to an external definition, as defined below, not an identifier declared with external linkage. An identifier may be declared with internal linkage and a function type. A definition for this function is only required (apart from the specified special case) if the identifier is used in an expression. If the result of the **sizeof** operator is an integer constant its evaluation can be performed during translation and there is no need for a definition to exist.

C++

The C++ Standard does not specify any particular linkage:

3.2p3 *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required.*

The definition of the term used (3.2p2) also excludes operands of the **sizeof** operator.

Other Languages

Some languages (e.g., Ada and Pascal) require a definition if there is a declaration of an identifier.

Coding Guidelines

Function declarations, with internal linkage, that are not referenced within a translation unit are redundant. The issue of redundant code is discussed elsewhere.

190 redundant code

Semantics

1814 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations.

Commentary

Clause 5.1.1.1 defines the term *translation unit* as such. The use of the term *external declarations* here refers to the syntactic definition. These external declarations include declarations of identifiers that have internal linkage.

110 translation unit known as
1810 external declaration syntax

C++

The C++ Standard does not make this observation.

1815 These are described as "external" because they appear outside any function (and hence have file scope).

Commentary

A term that is commonly used to refer to such declarations, by developers working in a variety of computer languages, is *global*.

C++

The C++ Standard does not refer to them as "external" in the syntax.

Coding Guidelines

Many developers intermix the terms *external* and *global*. There is no obvious benefit to be had in changing this practice (if this were at all possible).

1816 As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

Commentary

The discussion of this C behavior occurs elsewhere.

1353 definition identifier

1817 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. external definition

Commentary

This defines the term *external definition*. The definition of an inline definition specifies that it does not provide an external definition.

1541 inline definition not an external definition

C90

Support for inline definitions new in C99.

C++

The C++ Standard does not define the term *external definition*, or one equivalent to it.

Coding Guidelines

While many developers intermix the terms *external definition* and *external declaration*, the distinction between them is significant. Coding guideline documents need to ensure that they use correct terminology.

external linkage exactly one external definition

If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; 1818

Commentary

internal linkage 1813 exactly one external definition

This requirement is the external linkage equivalent of the requirement given for identifiers having internal linkage. However, violation of this requirement causes undefined defined behavior, it is not a constraint violation. The reason for this difference in behavior is that the C committee wanted implementations to be able to use third-party (e.g., the host OS vendor) linkers provided as part of the translation host environment. Some of these linkers support more relaxed, Fortran style, linkage conventions, and it is not possible to guarantee that a diagnostic will be issued for a violation of this requirement.

linkers 140

linkage 420

C90

Support for the **sizeof** operator having a result that is not a constant expression is new in C99.

C++

internal linkage 1813 exactly one external definition

The specification given in the C++ is discussed elsewhere.

Other Languages

Some languages (e.g., Ada) have sophisticated separate compilation mechanisms that require specialist linker support, while the designers of other languages have been willing to live within the limitations of linkers that their implementations are likely to make use of.

Common Implementations

Many linkers do not issue a diagnostic if more than one definition of the same identifier is contained within their input files. However, most linkers do issue a diagnostic if the program image they are asked to create includes a reference to an identifier for which there is no available definition.

Many linkers create identifiers for their own internal use. For instance some Unix linkers create the identifiers `etext`, `edata`, and `end`, to designate special addresses within a programs address space. These identifiers are used purely as symbols that represent an address, they occupy no storage. Some programs make use of the information provided by these addresses. Declarations, such as the following, can be used to provide an identifier that can be referenced in expressions (there is no definition of the object provided in the source of the program, it is provided by the linker).

```
1  extern const void end;
```

A fuller discussion of using such a declaration can be found in DR #012.

Coding Guidelines

identifier 422.1 declared in one file

Following the guideline recommendation dealing with the textual placement of all identifiers, having external linkage, in a single header prevents the creation of multiple declarations that may be incompatible with each other. However, every object or function referenced, in a program, requires a unique definition. One translation unit needs to contain an identifier's definition, while all the others only contain its declaration. There are a number of techniques for ensuring that declarations and their corresponding definition match. Two commonly seen techniques are:

1. using a macro, often called `EXTERN`, or `EXTERNAL`. For instance, consider a developer written header containing the following declaration:

   ```
   1  EXTERN int glob;
   ```

   In every translation unit, except one, that includes this header, the identifier `EXTERN` is defined as an object-like macro that expands to the keyword **extern**. In one translation unit the macro name expands to nothing, causing that declaration to become a definition. Variations on this technique are used to handle explicit initialization (in fact if an explicit initializer is given the absence or presence of the **extern** storage-class specifier is irrelevant, a definition is always created),

macro 1931 object-like

object 1354 reserve storage

2. placing a definition of the identifier, that is textually separate from the declaration in the header file, in one of the translation units. Use of this technique does violate the guideline recommendation specifying a single textual occurrence of a declaration (a definition is also a declaration). Recreating the problem of ensuring that both declarations are the same. 1353 definition identifier

In the case of functions, their declarations and definitions have to be textually separate. The complexities of using preprocessor directives to enable one textual occurrence to be used both as the declaration and the definition has a much higher cost than benefit. A method of ensuring that the two textual occurrences are the same is needed. One solution, that can be applied to both object and function declarations, is to make use of a property of the C language. It is possible to have multiple declarations of the same identifier, in the same scope, with external linkage, provided their declarations are compatible (a slightly less restrictive requirement than being the same). Translators are required to issue a diagnostic if the types are not compatible, so automated checking is performed.

Cg 1818.1

A source file that contains a textual definition of an object or function, having external linkage, shall **`#include`** the header file that contains its textual declaration.

1819 otherwise, there shall be no more than one.[137)]

Commentary

An identifier declared in a program may also be defined, even if it is not referenced in the program. However, whether such an identifier is referenced or not, the requirement on there being at most one definition is the same.

C++

The C++ Standard does not permit more than one definition in any translation unit (3.2p1). However, if a non-inline function or an object is not used in a program it does not prohibit more than one definition in the set of translation units making up that program.

Source developed using a C++ translator may contain multiple definitions of objects that are not referred.

Common Implementations

Some translators optimize away (by not writing any information about them to the object file) objects with internal linkage that are not referenced within the translation unit that contains their definition. Many linkers attempt to only include the definitions of objects and functions, in a program image, that are referenced from within that program.

Coding Guidelines

An identifier that is defined and not referenced is redundant code. This issue is discussed elsewhere. 190 redundant code

---

1820 137) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it. footnote 137

Commentary

This permission is needed to support the C model of separate translation. A header file containing many declarations may be included in the source files used to build a program. However, because a definition for an identifier declared in a header is only required if the identifier is referenced, developers do not have to be concerned about what is declared in the headers they include.

C++

The C++ Standard does not make this observation.

Other Languages

Whether or not it is possible to declare an identifier without also providing a definition depends on the separate translation model used by a language.

6.9.1 Function definitions

function definition syntax


```
function-definition:
          declaration-specifiers declarator declaration-list_opt compound-statement
declaration-list:
          declaration
          declaration-list declaration
```

Commentary

Syntactically it is not possible to tell the difference between a function definition and a function declaration until the first token after the declaration list is seen (which could be either a semicolon, an opening brace, or an identifier).

C++

The C++ Standard does not support the appearance of *declarator-list*$_{opt}$. Function declarations must always use prototypes. It also specifies additional syntax for *function-definition*. This syntax involves constructs that are not available in C.

Other Languages

Many other languages use a keyword to indicate that a body of code is being defined. The keywords used include **procedure**, **function**, and **subroutine**. Fortran supports the creation of statement functions within a subroutine. They essentially specify a form of parameterized expression, that can also access the objects visible at the point they are defined. They are invoked using the function call notation.

Common Implementations

An extension supported by `gcc` allows developers to specify attributes that a function definition possesses. For instance, some functions do not modify any objects that are not defined within their body. Making such information available to an optimizer means it does not have to make worst-case assumptions about the affects of a function call. The following declaration specifies that the function `square` has this attribute:

```
1  int square(int) __attribute__((const));
```

The IAR PICmicro compiler[612] supports the **`__monitor`** specifier, which causes machine code to be generated that ensures the specified function is executed as an atomic entity (i.e., interrupts are disabled during its execution).

value profiling 940

Recent research has shown that the value of some function parameters is the same over a large percentage of calls (for instance, based on using the SPEC95 dataset as input to `gcc`, in 70% of cases the value of the third parameter of calls to the function `simplify_binary_operation` was 34[986]). A cost/benefit analysis can be used to decide whether it is worthwhile creating a specialized version, optimized for a known value of one of the parameters, of a function. The cost of such specialization is the addition, by the translator, of a check against the common value to decide whether to jump to the specialized or unspecialized version. The benefit occurs when the specialized version executes much more quickly (flow analysis making use of the known parameter value to improve the quality of machine code generated for the specialized version). To be worthwhile the overall increase in performance in the specialized version has to be greater than the decrease in performance caused by the test against the frequently occurring value on function entry.

Coding Guidelines

A program usually contains more than one function. Splitting a program up into what are sometimes called *modules* (i.e., C functions) has a long history. A variety of reasons for creating and using functions have been proposed, including the following:

- Use of appropriately sized functions (not to big and not to small, sometimes known as the *Goldilocks principle*) is believed to have various benefits (although studies validating these claims appear to be non-existent[445]). Park[1054] discusses how statements might be counted, and Fenton[417] discusses using size metrics to predict the likely number of defects contained in software. While it is possible to plot measurements of various quantities against each other (e.g., a plot of defect density against function size produces a U-shaped curve, showing that very small functions and very large functions contain more defects per line of code than medium sized functions[550]), the measurements are usually averaged over all functions in a program and causal links between the two quantities are rarely established. One problem with measurements based on the size of individual functions is that it is possible artificially change the results obtained, by splitting one function into many, or merging several into one function. For instance, reducing the number of control flow paths[1447] through a function is often seen as beneficial, because it reduces the amount of path testing that needs to be performed. A reduction in the number of paths in any function definition can be achieved by breaking it up into smaller functions. However, the reduction achieved is an artifact of the measuring process, which is based on individual definitions. The number of paths through the program has not been reduced.

- It is part of the culture of writing source code (it's what students are told they must do when learning to program, and developers who don't split their code into appropriately sized functions are often chastised for this behavior). Various program designed methodologies have specified rules for how a program should be decomposed. For instance, the structured design movement (the original Stevens, Myers, and Constantine paper was recently republished[1296]) used the idea of coupling and cohesion between statements as a means of deciding which of them belonged in a module. There have been proposals to define the concepts of coupling[1027] and cohesion[993] in more mathematical terms. However, while this formalism enables calculations to be made by automatic tools, studies of applicability of these definitions to human readers are lacking (see Halliday and Hasan[536] for a discussion of cohesion in English text). Concept analysis is a technique for identifying groupings of objects, in existing source code, that share common attributes. It has been used identify possible C++ classes in C source code.[1235]

  coupling and cohesion

  concept analysis

- It enables the available work to be distributed to be distributed across more than one person.

- Reducing the amount of duplicate, or very similar, code in a program. This can reduce maintenance costs by minimizing the amount of existing code that needs to be modified when changes to the behavior of a program are made (measurements of duplicate code remaining in existing programs are discussed elsewhere).

  1821 duplicate code

- Splitting a program into independent modules makes it easier to change parts of it in the future. The flexibility offered by the ability to upgrade parts of a hardware system, as technology improved, rather than having to buy a new system has been shown to offer significant economic advantages for both vendors and users.[87] Breaking a system up into modules is not enough, it is also necessary to hide information; to be exact, information that is likely to change has to be hidden within individual modules.[1059] Predicting in advance which parts of a program are likely to be changed is a difficult problem. One proposed solution is based on the economic theory of options,[1312] valuing software structures on the basis of the cost of changing them in the future.

- Reuse of software once written (e.g., libraries of functions). This issue is not considered in these coding guidelines.

A large part of comprehending a program involves comprehending the side effects generated from executing the statements contained in particular function definitions.[1821.1] A developer's ability to predict the behavior of a program is based on their knowledge of the behavior of individual function definitions (e.g., how they modify data structures and which other functions they might call). A question often asked, by developers, about a function is "what does it do?". In many ways a function definition represents an episode of a program, which when executed in sequence with other functions tell the story that is program execution.

statement 1707 syntax

The coding guideline discussion on statements drew a parallel with studies of human sentence processing, in an effort make use of some of the findings of those studies. The discussion in of function definitions, in this coding guideline subsection, extends the use of this parallel to studies of story comprehension. It is assumed that existing human story telling and comprehension skills are something developers apply to the reading and writing of code.

A function definition is the largest unit of contiguous source code that readers might generally expect to read from start to finish.[1821.2] Reading the story of a programs execution invariably requires readers to look at a variety of different functions which are not usually visually close to the text of the definition current being read. This behavior differs from that required to read the written form of prose stories, where the intended narrative flows sequentially through visually adjacent text (although readers might only read a few chapters at a time).

memory 0 episodic
memory 0 semantic

Developer memory for a function definition may depend on when they last read it. For instance, a definition that has just been read might be stored in episodic memory, while one that was read some a few days ago might be stored in semantic memory. Whether or not these differences in human memory storage mechanism, for source code information, affects developer performance is not known.

Adults have an extensive knowledge of routine activities (e.g., eating in a restaurant or going shopping) and spend a significant amount of their time performing these activities (they are very practiced at performing some of them). The *script theory* of Schank and Abelson[1218] proposes that part of a person's knowledge is organized around a large number of sterotypical situations involving activities that are often performed.

Studies of the structure of narrative stories have a long history, while the study of the structure of source code is still in its infancy. The following illustrates some of the studies that have investigated regularity and repeated elements in prose stories and source code:

- A study by Mandler and Johnson[894] analyzed the structure of simple stories and created a grammar to describe it. In the following grammar STATE may be external (e.g., a current conditional of the world) or internal (e.g., an emotional state of mind). An EVENT is any occurrence or happening and may also be external or internal. The terminals AND, THEN, and CAUSE represent various relationships that may connect states and events.

```
STORY      -> SETTING AND EVENT_STRUCTURE
SETTING    -> STATE* (AND EVENT*) | EVENT*
STATE*     -> STATE ((AND STATE)^N)
EVENT*     -> EVENT (((AND | THEN | CAUSE) EVENT)^N)((AND STATE)^N)
EVENT_STRUCTURE -> EPISODE ((THEN EPISODE)^N)
EPISODE    -> BEGINNING CAUSE DEVELOPMENT CAUSE ENDING
BEGINNING -> EVENT* | EPISODE
DEVELOPMENT       -> SIMPLE_REACTION CAUSE ACTION | COMPLEX_REACTION CAUSE GOAL_PATH
SIMPLE_REACTION  -> INTERNAL_EVENT ((CAUSE INTERNAL_EVENT)^N)
ACTION     -> EVENT
COMPLEX_REACTION -> SIMPLE_REACTION CAUSE GOAL
GOAL       -> INTERNAL_STATE
GOAL_PATH -> ATTEMPT CAUSE OUTCOME | GOAL_PATH (CAUSE GOAL_PATH)^N
ATTEMPT    -> EVENT*
```

[1821.1] Many source code metrics are based purely on counting some attribute of the contents of function definitions.
[1821.2] In object-oriented languages the largest such unit might sometimes be a class.

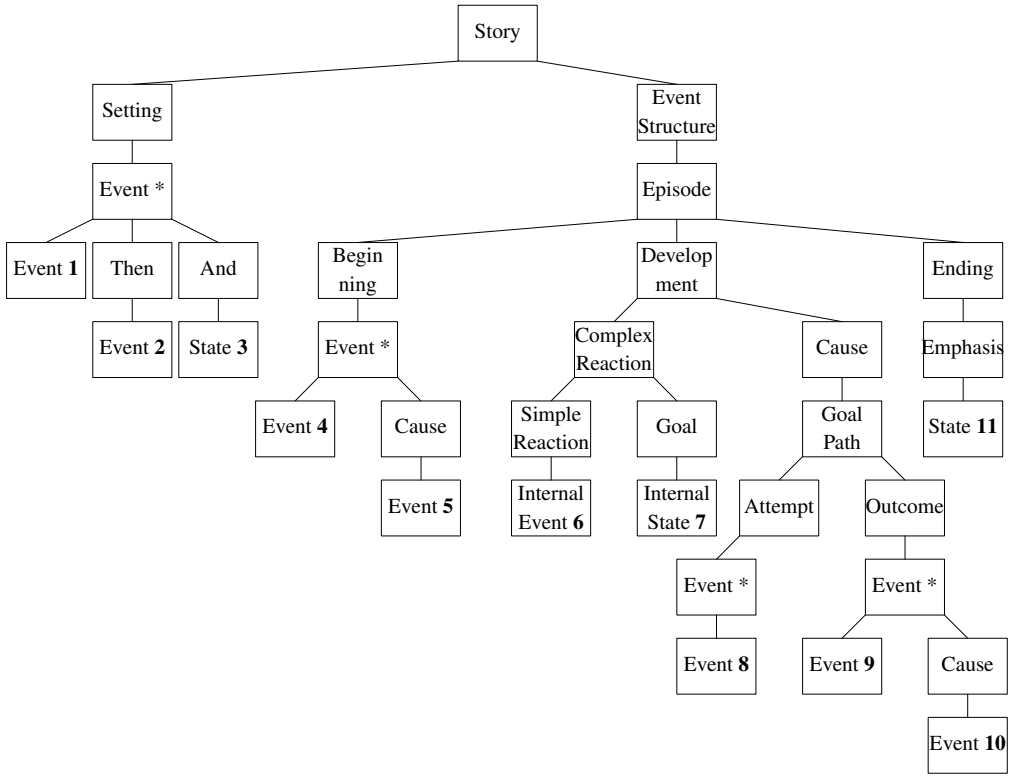


Figure 1821.1: Parse, using the Story grammar, of the tale of a dog and piece of meat. Adapted from Mandler and Johnson.[894]

```
OUTCOME   -> EVENT* | EPISODE
ENDING    -> EVENT* (AND EMPHASIS) | EMPHASIS | EPISODE
EMPHASIS  -> STATE
```

The following is an example of a story following this grammar:[894]

```
1  It happened that a dog had got a piece of meat
2  and was carrying it home in his mouth.
3  Now on his way home he had to cross a plank lying across a stream.
4  As he crossed he looked down
5  and saw his own shadow reflected in the water beneath.
6  Thinking it was another dog and another piece of meat,
7  he made up his mind to have that also.
8  So he made a snap at the shadow,
9  but as he opened his mouth the piece of meat fell out,
10 dropped into the water,
11 and was never seen again.
```

- Most attempts to find semantic narrative in software have been based on what are called *programming plans*. Global plans being built in a bottom-up fashion from a database of known local plans. A local plan is based on the control and data flow constructs of a group of statements (e.g., a loop over the elements of an array that performs some action when one of the elements matches a fixed value may match a linear search plan) (Hartman[546] discusses the recognition of local plans based on concepts from a number of domains, including the application domain, mathematics, and known programming techniques; Wills[1470] describes building a database of clichés and searching for them in a graphical

representation of the program). The problem of recognizing a particular program plan in source code is NP-hard[1483] ($O(S^A)$, where S is the size of the program, measured in units matched by subplans, and A is the number of subplans within the plan). Various matching heuristics have been proposed, including constraint based methods Woods.[1482] Once detected plans (clichés, concepts, schemas, templates, or some other term) have been used to locate and correct common novice programmer mistakes,[676] recognize algorithms that can be transformed into a more efficient form (e.g., matrix multiply).[352]

When presented with a sequence of actions peoples attempt to match them against patterns of action they are familiar with. A brief description, providing an overview, can have a significant impact on comprehension and recall performance (this issue is discussed elsewhere).

identifier 792
cue for recall

A persons experience with the form and structure of these repeated elements seems to be used to organize the longer-term memories they form about them. The following studies illustrate the affect that peoples knowledge of the world can have on their memory for what they have read, particular with the passage of time, and their performance in interpreting sequences of related facts they are presented with:

- A study by Bower, Black, and Turner[146] gave subjects a number of short stories describing various activities (i.e., scripts), such as visiting the dentist, attending a class lecture, going to a birthday party, etc., to read. Each story contained about 20 actions, such as looking at a dental poster, having teeth x-rayed, etc. There was then a 20 minute interval, after which they were asked to recall actions contained in the stories. The results showed that subjects around a quarter of recalled actions might be part of the script, but that were not included in the written story. Approximately seven percent of recalled actions were not in the story and would not be thought to belong to the script. A second experiment involved subjects reading a list of actions which, in the real world, would either be expected to occur in a known order or be not expected to have any order (e.g., the order of the floats in a parade). The results showed that, within ordered scripts, actions that occurred at their expected location were recalled 50% of the time while actions occurring at unexpected locations were recalled 18% of the time at that location. The recall rate for unordered scripts (i.e., the controls) was 30%.
- A study by Graesser, Woll, Kowalski, and Smith[511] read subjects stories representing scripted activities (e.g., eating at a restaurant). The stories contained actions that varied in the degree to which they were typical of the script (e.g., Jack sat down at the table, Jack confirmed his reservation, and Jack put a pen in his pocket).

Table 1821.1: Probability of subjects recalling or recognizing typical or atypical actions present in stories read to them, at two time intervals (30 minutes and 1 week) after hearing them. Based on Graesser, Woll, Kowalski, and Smith.[511]

Memory Test	Typical (30 mins)	Atypical (30 mins)	Typical (1 week)	Atypical (1 week)
Recall (correct)	0.34	0.32	0.21	0.04
Recall (incorrect)	0.17	0.00	0.15	0.00
Recognition (correct)	0.79	0.79	0.80	0.60
Recognition (incorrect)	0.59	0.11	0.69	0.26

The results showed (see Table 1821.1) that recall was not affected by typicality over short periods of time, but that after one week recall of atypical actions dropped significantly. Recognition (i.e., subjects were asked if a particular action occurred in the story) performance for typical vs. atypical actions was less affected by the passage of time.

- A study by Dooling and Christiaansen[366] asked subjects to read a short biography containing 10 sentences. The only difference between the biographies read by subjects was that in some cases the name of the character was fictitious (i.e., a made up name), while in other cases it was the name of an applicable famous person. For instance, one biography described a ruthless dictator and used

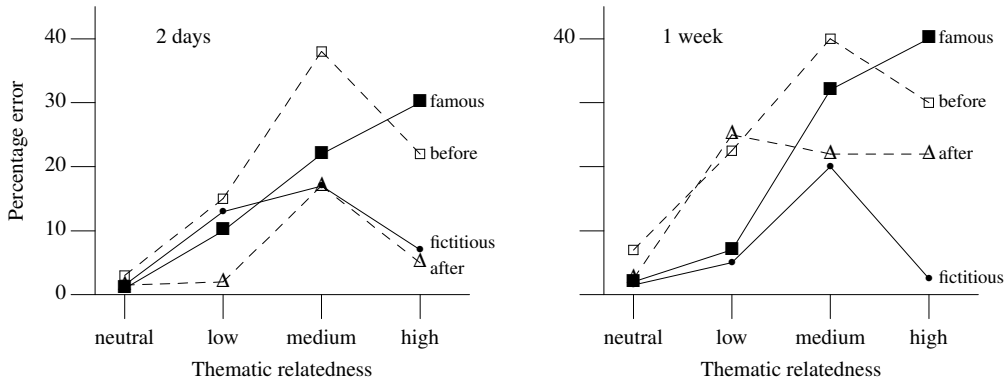


Figure 1821.2: Percentage of false-positive recognition errors for biographies having varying degrees of thematic relatedness to the famous person, in *before*, *after*, *famous*, and *fictitious* groups. Based on Dooling and Christiaansen.[366]

either the name Gerald Martin or Adolph Hitler. After a period of two days, and then after one week, subjects were given a list of 14 sentences (seven sentences that were included in the biography they had previously read and seven that were not included) and asked to specify which sentences they had previously read. To measure the impact of subjects' knowledge about the famous person, on recognition performance, some subjects were given additional information. In both cases the additional information was given to the subjects who had read the biography containing the made up name (e.g., Gerald Martin). The *before* subjects were told just before reading the biography that it was actually a description of a famous person and given that persons name (e.g., Adolph Hitler). The *after* subjects were told just before performing the recognition test (they were given one minute to think about what they had been told) that the biography was actually a description of a famous person and given that persons name.

The results (see Figure 1821.2) are consistent with the idea that remembering is constructive. After a week subjects memory for specific information in the passage is lost. Under these conditions recognition of sentences is guided by subjects general knowledge. Variations in the time between reading the biography and identity of a famous character being revealed affected the extent to which subjects integrated this information.

- A study by Kintsch, Mandel, and Kozminsky[737] measured the time taken to read and summarize 1,400 word stories. In the text seen by some students the order of the paragraphs (not the sentences) forming the story were randomized. The results showed that while it was not possible to distinguish between the summaries produced by subjects reading ordered vs. randomised stories, reading time for random paragraph ordering was significantly longer (9.05 minutes vs. 7.34).

Developers do not read the source code of a function definition every time they encounter a reference to it. If it has been read before they may be able to recall sufficient information about it to satisfy their immediate needs (how people trade-off the costs of recalling knowledge in their heads against obtaining knowledge from the real world is discussed elsewhere). The results of studies of story recall performance suggest that recall of events is more accurate when the events that occur correspond to reader expectations (which might be based on their knowledge of program plans, or application domain knowledge) about what events should occur.

0 cost/accuracy trade-off

Developers use their knowledge of what functions do in several ways, including:

- to deduce the expected affects of performing a call to that function,
- to work out which functions need to be updated when modifying the behavior of a program, and
- to work out which functions may be affected by changes to the definition of other functions,

In all cases forgetting about an action performed by the function definition or assuming that it performs some action, when it does not it, can result in unexpected behavior during program execution.

The expectations that a reader might have about the actions performed by particular function definitions may require knowledge of the application domain, algorithms, C idioms, and development group conventions. The analysis of what actions do, or don't, meet these expectations can only be carried out (for the time being) by developers familiar with these domains. Also it might not be possible to simultaneously satisfy expectations from all of these domains, it is likely that trade-offs will need to be made.

An example of an action carried out within a function definition that would be surprising, or unexpected, might be the disabling of interrupts within a function that performs block copies of storage (perhaps special registers used by the operating system are being used to speed up the copy operation, which are restored once the copy has completed).

The parameters in a function definition are likely to be read more often that the parameters in any declaration of it. For this reason it is likely to be worth investing more in laying out the visible form of parameter declarations. The issue of declaration layout is discussed elsewhere.

declaration 1348
visual layout

mixing declarations and statements

Order of declarations and statement

Developers often have some flexibility in how they order declarations and statements within a function. For instance, one or more of the following patterns of usage are often seen in source:

- The declarations for all of the locally defined objects occur at the start of the function (it is rare to see a compound block opened and closed around a short sequence of statements purely to declare an object whose required lifetime and scope is limited to those statements) (see Figure 408.1).
- Grouping statements by the action they perform. For instance, initializing all objects at the start of a function, or performing all output at the end of a function.
- Grouping statements that access the same objects together. For instance, initializing objects close to where they are first accessed, or performing output as soon as the necessary values are known.

There are a number of reasons why grouping statements by the objects they access might offer greater benefits than other grouping algorithms, including:

- Minimizing the number of separate sequences of source that need to be cut-and-pasted when copying or reorganizing code.
- Minimizing the amount of source that intervenes between two statements containing information that needs to be integrated (into a model of function behavior) increases the probability that readers will make the correct inferences.

statements 1707
integrating information between

Recommending that related sequences of statements and/or declarations be grouped together begs the question of how to measure *related*. Given the current lack of an algorithmic measure the only alternative is to fall back on developer preferences (e.g., code reviews).

Rev 1821.1 Where possible, statements that are related to each other shall be visible close to each other in the source code.

Some coding guideline documents recommend a maximum number of lines, or statements, in a function definition. However, arbitrarily splitting a function into smaller functions purely to meet some recommended limit (experience suggests that some developers do arbitrarily split function that are considered to be too large; adherence to guideline recommendations cannot force developers to think deeply about what they are doing).

Knowing when to split a large function into smaller functions depends on developer ability to spot separate concepts. It is a matter of experience.

One practical consideration is the finite number of source lines that can be viewed on a display device at any time. Being able to view the complete source of a function removes the need for any external effort apart from eye movements (developers trading off the costs of physical and mental effort, needed to obtain information, is discussed elsewhere). It may, or may not, be possible to comprehend a function by reading its visible source without referring back to previously read material that are no longer visible on the display.

0 cost/accuracy trade-off

Functions definitions do not always increase in size. For instance, duplicate sequences of code, appearing in different parts of the same or different functions, may suggest the creation of a function containing a single instance of that code (this process reduces the size of existing functions).

Duplicate source code

duplicate code

The same code may occur in several places of the same program (the source does not have to be character for character identical to be considered *the same*, commenting or layout may differ, some identifiers may have different spellings, or one or more lines may have been added or deleted). The terms *duplicate code* or *clones* is often used to describe instances of similar code.

The following are some of the reasons why duplicate code may exist in a program:

- The functionality required is very similar to that implemented by some existing source. The developer copies this existing source to use as a template for the new functionality, which may only involve a few changes to the original.
- Enhancing performance. For instance, replacing a function call, within a loop, by a copy of the body of the function.
- C's lack of support for generic types. For instance, a function defined to perform some operation on objects of type `X` and a duplicate definition that performs the same operation on objects having type `Y`. One solution here is to use call backs (e.g., the `qsort` library function has no need of any knowledge of the type of the objects being compared because it calls a developer supplied function to perform comparisons).
- Oversight. On large development efforts it is not unknown for two developers to implement functions having the same functionality.
- Coincidence. In any large program there are likely to be sequences of statements that are very similar, even though their purpose is completely different.

Duplication of source code not only increases the size of programs, it also increases the possibility of faults being created by modifications to existing code.[667] For instance, when one sequence of statements is modified, but a duplicate sequence in another part of the program is left unmodified (the unmodified statements may not have needed modification; however, experience suggests that there is a very real likelihood that they did).

Merging duplicate sequences of statements into a single textual occurrence ensures that any changes that need to be made only need to be made in one place. Whether this one place is a function (inline or not) or a macro is a developer decision.

0 agenda effects decision making

It is to be expected that programs will contain individual lines that are identical. Similarity only becomes noteworthy when significant amounts of code have a high degree of similarity. Judgments of what constitutes significant can vary. Detecting duplicate code can be a computationally expensive process and a number of different algorithms have been proposed, including:

- Baker[85] built a C based tool `dup` that looked for either exact matches (ignoring white space) or parameterized matches (called *p-matches*, where the spelling of identifiers and constant literals could be different). Running `dup` on 714,479 lines of the X Window System it found 2,487 matches (representing 976 groups, each instances of code that had been copied and edited) of at least 30 lines (representing 19% of the code).

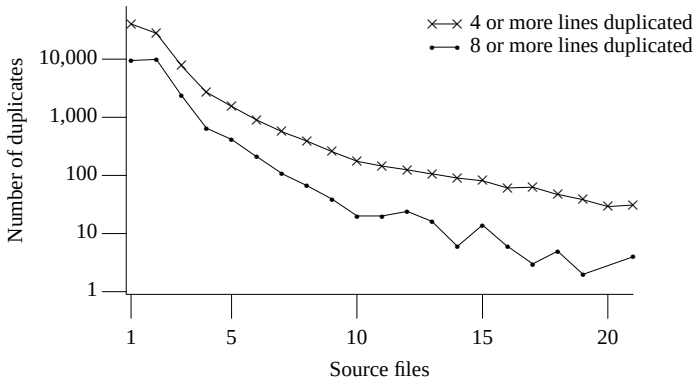


Figure 1821.3: Number of instances of duplicate physical lines, where a given duplicate line sequence is contained within a single source file or more than one source file (ignoring comments and blank lines) for sequences having at least 4 and 8 lines. Data created by processing the `.c` files (for each of the book's program's complete source tree) using `Simian`.[1151]

Table 1821.2: Number of clones (the same sequence of 30 or more tokens, with all identifiers treated as equivalent) detected by CCFinder between three different operating systems (Linux, FreeBSD, and NetBSD). Adapted from Kamiya, Kusumoto, and Inoue.[711]

O/S pairs	Number of Clone Pairs	% of Lines Included in a Clone	% of Files Containing a Clone
FreeBSD/Linux	1,091	FreeBSD (0.8) Linux (0.9)	FreeBSD (3.1) Linux (4.6)
FreeBSD/NetBSD	25,621	FreeBSD (18.6) NetBSD (15.2)	FreeBSD (40.1) NetBSD (36.1)
Linux/NetBSD	1,000	Linux (0.6) NetBSD (0.6)	Linux (3.3) NetBSD (2.1)

- Code that has been cut-and-pasted may subsequently have small changes made to it. A number of researchers have attempted to detect plagiarism,[1120, 1417] where an attempt has been made to hide the origins of the code (e.g., in student assignments).
- It is often possible for different sequences of the same set of statements to have the same effect. However, the dependencies between statements will not be affected by any differences in ordering and comparisons based on a dependence graph will be able to find duplicates.[773]
- Other studies have used similarity measures based on the abstract syntax tree,[102] software metrics,[907] and the visible source (after white space and comments had been removed).[370]

Analysis of programs, that have been developed over a period of time, shows that they contain a surprisingly large amount of duplicate code, and that the percentage of clone functions stays approximately the same over multiple releases of a product (6–8% clones over 6 product releases as the number of function grew from 170,00 to 206,000[791]).

Not all applications are made up of a single program. For large applications it can be worthwhile to have a number of smaller programs, each performing a specific function. Once the decision is made to have separate programs there is the real possibility that source code will be cut-and-pasted between different programs, rather than a common set of library functions created.

A study by Tonella, Antoniol, Fiutem, and Calzolari[1355] describes an application containing 4.7 M lines of code making up 402 programs (there were that many functions called main) linking against 815 compiled translation units (libraries). A metric taxonomy[907] was used to detect function clones. Out of 7,277 functions; 1,016 had the same name and metric values, 609 had different names but the same metric

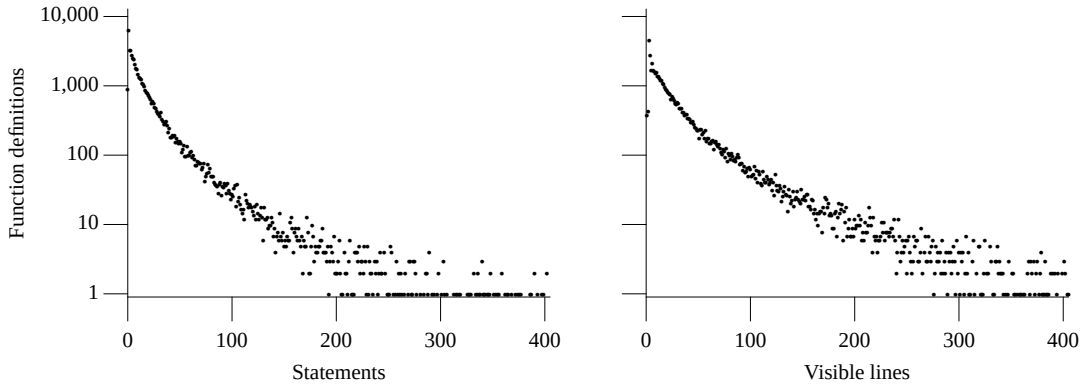


Figure 1821.4: Number of function definitions containing a given number of statements and visible source lines. Based on the translated form of this book's benchmark programs.

values, and 621 functions differed by a single metric. No results were given for how many functions were actually removed from the complete application suite. They were under considerable time pressure which meant it was not possible to consider all function clones.

Use of functions invariably causes developers to consider efficiency issues. The efficiency issues associated with the call/return overhead are discussed elsewhere as are the issue of passing parameters versus using global objects.

1004 register function call housekeeping
288 limit parameters in definition

Usage

A study of over 3,000 C functions by Harrold, Jones, and Rothermel[545] found that the size of a functions control dependency graph was linear in the number of statements (the theoretical worst-case is quadratic in the number of statements).

A study by Neamtiu, Foster, and Hicks[998] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 81% of releases one or more existing function definitions had their argument signature changed, while one or more function definitions had their return type changed in 42% of releases and one or more function definitions had their name changed in 49% of releases.[997]

Table 1821.3: Static count of number of functions and uncalled functions in SPECint95. Adapted from Cheng.[222]

Benchmark	Lines of Code	Number of Functions	Uncalled Functions	Benchmark	Lines of Code	Number of Functions	Uncalled Functions
008.espresso	14,838	361	46	126.gcc	205,583	2,019	187
023.eqntott	12,053	62	2	130.li	7,597	357	1
072.sc	8,639	179	8	132.ijpeg	29,290	477	16
085.cc1	90,857	1,452	51	134.perl	26,874	276	13
124.m88ksim	19,092	252	13	147.vortex	67,205	923	295

How many instructions are executed, on average, in a function definition? It will depend on the characteristics of the translator and host processor (see Table 1821.4).

0 translation technology

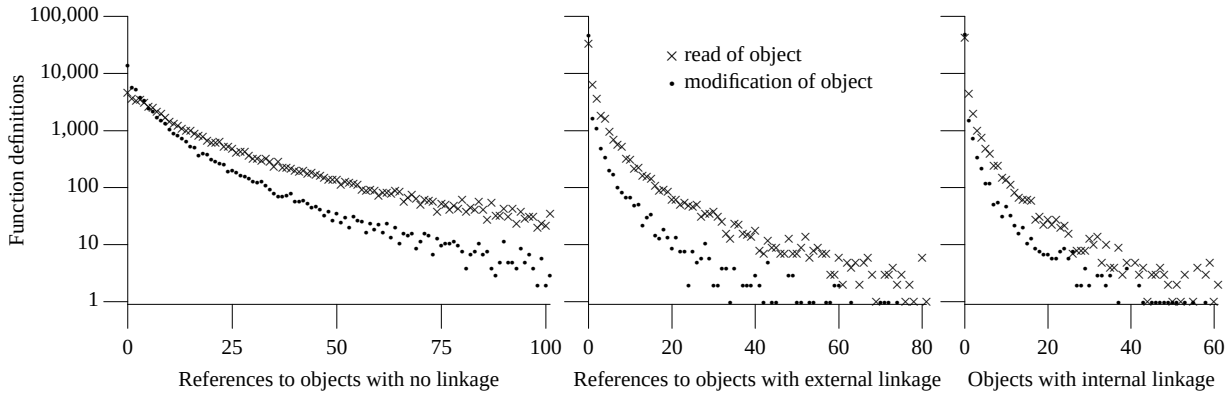


Figure 1821.5: Number of function definitions containing a given number of references (i.e., an access or modification) to all objects, having various kinds of linkage. Based on the translated form of this book's benchmark programs.

Table 1821.4: Mean number of instructions executed per function invocation. Based on Calder, Grunwald, and Zorn.[192]

Program	Mean	Leaf	Non-Leaf	Program	Mean	Leaf	Non-Leaf
burg	61.6	30.6	142.8	eqntott	386.8	402.8	294.2
ditroff	58.6	72.3	56.3	espresso	244.9	151.3	526.5
tex	173.2	44.3	205.4	gcc	96.4	30.1	123.5
xfig	61.9	38.6	74.8	li	42.5	31.9	44.2
xtex	114.9	93.9	136.5	sc	71.1	49.4	80.1
compress	368.4	1,360.2	367.5	Mean	152.8	209.6	186.5

Table 1821.5 gives a breakdown of the overall control flow characteristics of function bodies. One explanation for the larger number of SPECint benchmark functions containing iteration statements is that these programs were selected on the basis of primarily being cpu bound. The only practical way of using lots of cpu time is to iterate and hence this benchmark is biased in favour functions that iterate a lot.

Table 1821.5: Contents of function bodies (as a percentage of all bodies) for embedded `.c` source,[390] SPECint95, and the translated form of this book's benchmark programs.

	Embedded	SPECint95	Book benchmarks
Trivial (one basic block)	32.7	16.2	57.1
Non-looping	47.9	48.1	18.1
Looping	19.4	35.7	24.8

Usage information on the number of objects defined within a function definition is given elsewhere (see Figure 286.1).

Constraints

The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[138)] 1822

Commentary

declarator syntax 1547 The syntax for function definitions requires a declarator, which means that the following:

```
1 int x
2 { }
```

is syntactically valid. However, it violates the above constraint. In the following declarations:

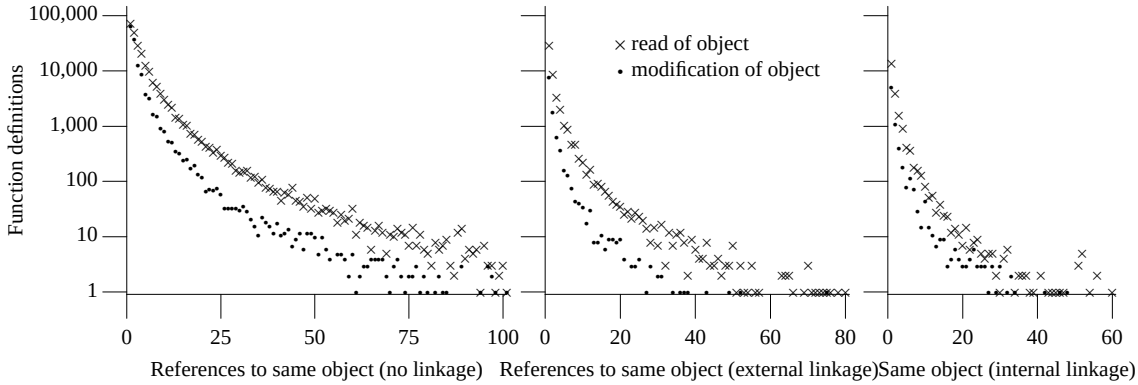


Figure 1821.6: Number of function definitions containing a given number of references (i.e., an access or modification) to the same object, having various kinds of linkage. Based on the translated form of this book's benchmark programs.

```
1  typedef int F(void);
2  F f { /* ... */ }
```

the declarator `f` does not syntactically specify a function type (see the footnote for further discussion). 1830 footnote 138

C++

The C++ Standard specifies the syntax (which avoids the need for a footnote like that given in the C Standard):

The `declarator` *in a* `function-definition` *shall have the form* 8.4p1

D1 (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

Common Implementations

Many translators use the requirements specified in this constraint to simplify the language grammar they need to process. Consequently the diagnostic message they issue for a violation of this constraint often refers to a violation of syntax.

1823 The return type of a function shall be **`void`** or an object type other than array type. function definition return type

Commentary

This wording is slightly different from that specified for function declarators, but the set of return types supported is the same. While function types are not included, pointers to functions are object types and are therefore permitted. 1592 function declarator return type

Array types are converted to pointer types in many contexts. Given the occurrence of these implicit conversions, writing an expression (e.g., in a **`return`** statement) that has an array type is relatively involved. A special case dealing with objects having an array type in a **`return`** statement would add complications to the language for little obvious benefit (it is possible to declare a function to return a structure type, which has a member having an array type). 729 array converted to pointer

C++

Types shall not be defined in return or parameter types. 8.3.5p6

The following example would cause a C++ translator to issue a diagnostic.

```
enum E {E1, E2} f (void) /* does not change the conformance status of program */
                         // ill-formed
{
return E1;
}
```

Other Languages

A number of languages support functions returning array types and some languages (e.g., those in the functional family of languages) support functions returning function types or even partially evaluated functions. However, some languages further restrict the return type to being a scalar type.

Example

```
int (*g)(void)    /* Constraint violation. */
{ /* ... */ }

int (*h(void))[2] /* Constraint violation. */
{ /* ... * }
```

Usage

Usage information on function return types in the `.c` files is given elsewhere (see Table 1005.1).

Table 1823.1: Occurrence of function return types (as a percentage of all return types; signedness and number of bits appearing in value representation form) appearing in the source of embedded applications (5,597 function definitions) and the SPECint95 benchmark (2,713 function definitions). A likely explanation of the greater use of type **`void`** is the perceived performance issues associated with returning values via the stack causing developers to return values via objects at file scope. Adapted from Engblom.[390]

Type/Representation	Embedded	SPECint95	Type/Representation	Embedded	SPECint95
`void`	59.4	31.2	ptr-to . . .	2.0	17.1
unsigned 32 bit	0.5	2.2	signed 32 bit	0.3	48.4
unsigned 16 bit	3.3	0.0	signed 16 bit	1.6	0.2
unsigned 8 bit	31.6	0.5	signed 8 bit	0.8	0.0

The storage-class specifier, if any, in the declaration specifiers shall be either **`extern`** or **`static`**. 1824

Commentary

In the context of function declarations, storage-class specifiers are used to specify linkage. The two chosen where **`extern`** and **`static`** (representing external and internal linkage respectively), making the appearance of any other storage-class specifier meaningless.

C++

7.1.1p2 *The **`auto`** or **`register`** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).*

A C++ translator is not required to issue a diagnostic if these storage-class specifiers appear in other contexts. Source developed using a C++ translator may contain constraint violations if processed by a C translator.

Common Implementations

One possible interpretation that could be given to the **`register`** storage-class appearing in a function definition, is as a hint to translators that the machine code for the function body be kept in a processor's cache 0 instruction cache. However, support for explicit program control of the cache is only just starting to appear in commercially available processors.

Coding Guidelines

The coding guideline issues associated with function declarations that include the storage-class specifier **static**, **extern**, or no storage-class are discussed elsewhere.

425 static internal linkage
426 extern identifier linkage same as prior declaration
430 function no storage-class

1825 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier.

Commentary

The syntax for a parameter type list in a function definition is the same as that used for function declarations, where no identifiers need be specified. However, in the case of a function identifiers are required, otherwise there is no mechanism for accessing any argument value passed in that position (use of the ellipsis does not involve the explicit specification of type information)

1601 ellipsis supplies no information

C++

An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called "formal argument"). [Note: in particular, parameter names are also optional in function definitions and . . .

8.3.5p8

[Note: unused parameters need not be named. For example,

```
void print(int a, int)
{
    printf("a = %d\n",a);
}
```

—end note]

8.4p5

Source developed using a C++ translator may contain unnamed parameters, which will cause a constraint violation if processed by a C translator.

1826 No declaration list shall follow.

Commentary

A declaration list can only follow when the form of the function declarator is the old style.

1827 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared.

identifier list declare at least one declarator

Commentary

The declaration list declares the identifiers that appear in the parameter list and only those identifiers. Standard support for *old style* function definitions was needed because of the existence of a large body source code containing them. This constraint does not prohibit some declarations of doubtful utility. For instance:

```
1  void f_1(p)
2  enum {x, y      /* x & y appear in a declaration-specifier, not in a declarator list. */
3            } p;
4  { /* ... */ }
5
6  void f_2(p)
```

```
7   struct {
8           int z;  /* z is declared, but is not in the declarator list. */
9          } *p;
10  { /* ... */ }
```

C90

The requirement that every identifier in the identifier list shall be declared is new in C99. In C90 undeclared identifiers defaulted to having type **int**.

Source files that translated without a diagnostic being issued by a C90 translator may now result in a diagnostic being generated by a C99 translator.

C++

The declaration list form of function definitions is not supported in C++.

Common Implementations

It is likely that many C99 implementations will offer a C90 compatibility option, that will successfully translate source files containing functions whose parameter definition includes identifiers that are not explicitly declared in the declaration list.

Example

Fortran allows function parameters to be referred to before they are defined and it is established practice to declare an array parameter before the length specified in its array bounds. Translating such Fortran source into C required either that the parameter order be reversed, or old style definitions be used.

```
1   extern void f_proto(int, double [*][*]);
2   extern void f_old();
3
4   /*
5    * C requires identifiers to be declared before they are referenced.
6    */
7   void f_proto(int p_length, double p_1[p_length][p_length])
8   { /* ... */ }
9
10  void f_old(p_1, p_length) /* Parameter order follows Fortran conventions. */
11  int p_length;
12  double p_1[p_length][p_length];
13  { /* ... */ }
14
15  /*
16   * Hanging on to a bit more type information...
17   */
18  void f_alternative(double *p_1, int p_length)
19  {
20  double (*loc_p_1)[p_length] = (double (*)[p_length])p_1;
21  }
```

An identifier declared as a typedef name shall not be redeclared as a parameter. 1828

Commentary

Traditionally C source has been syntactically processed using a parser that only examined the next token in the input stream (i.e., the grammar can be processed using a LALR(1) tool, such as `yacc` and `bison`). This C constraint means that translators are not required to process the following code:

```
1   typedef int I;
2
3   int f(I) /* Constraint violation. */
4   int I;
5   { /* ... */ }
```

which requires more than one token lookahead to answer the question: is `f` an old style function talking a parameter called `I`, or is `f` a declaration of a prototype taking a parameter of type **`int`**, with no identifier name given?

C++

The form of function definition that this requirement applies to is not supported in C++.

1829 The declarations in the declaration list shall contain no storage-class specifier other than **`register`** and no initializations.

function declaration list storage-class specifier

Commentary

Parameters have automatic storage duration and are initialized by the value of the corresponding argument.

1838 parameter automatic storage duration
1004 function call preparing for

C++

The form of function definition that this requirement applies to (i.e., old-style) is not supported in C++.

Other Languages

Those languages (e.g., Ada) that support the use of default values on parameters, as a method of providing default values, usually require that they appear in the function declaration rather than its definition (so they are visible at the points in the source where calls to them occur).

1830 138) The intent is that the type category in a function definition cannot be inherited from a typedef:

footnote 138

```
typedef int F(void);            // type F is "function with no parameters
                                //               returning int"
F f, g;                         // f and g both have type compatible with F
F f { /* ... */ }               // WRONG: syntax/constraint error
F g() { /* ... */ }             // WRONG: declares that g returns a function
int f(void) { /* ... */ }       // RIGHT: f has type compatible with F
int g() { /* ... */ }           // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }        // e returns a pointer to a function
F *((e))(void) { /* ... */ }    // same: parentheses irrelevant
int (*fp)(void);                // fp points to a function that has type F
F *Fp;                          // Fp points to a function that has type F
```

Commentary

This requirement significantly simplifies the creation of grammars that can be processed by commonly available parser generators without any syntactic production rule conflicts occurring.

C++

The C++ Standard specifies this as a requirement in the body of the standard (8.3.5p7).

Semantics

1831 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters.

Commentary

Saying in words what is specified in the syntax. The return type is given by some of the *`declaration-specifiers`* and parts of the *`declarator`*.

C++

The C++ Standard does not explicitly make this association about function definitions (8.4).

Usage

Information on argument types is given elsewhere (see Table 1003.1).

Table 1831.1: Occurrence of parameter types in function definitions (as a percentage of the parameters in all function definitions). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
`struct *`	44.4	`void *`	3.4	`long`	1.6	`struct * *`	1.2
`int`	14.7	`union *`	3.1	`int *`	1.5	`enum`	1.2
other-types	6.8	`unsigned long`	2.7	`unsigned char *`	1.4	`const char *`	1.1
`unsigned int`	5.1	`unsigned int *`	2.0	`char * *`	1.3	`long *`	1.0
`char *`	4.7	`unsigned char`	1.6	`unsigned short`	1.2		

If the declarator includes a parameter type list, the list also specifies the types of all the parameters; 1832

Commentary

Each parameter has the type specified in this list, not the type of the corresponding parameter of any composite type (that may have been created because of the occurrence of previous declarations).

C++

If the parameter list is empty the C++ Standard defines the function as taking no arguments (8.3.5p2).

Example

```
1  extern int glob;
2  extern void f(int);
3
4  void f(const int p)
5  { /* p has type const int in this function body. */ }
6
7  void g(void)
8  {
9  f(glob); /* Composite type of f is function taking a parameter of type int. */
10 }
```

function prototype

such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. 1833

Commentary

compos-ite type 642

If there are any previous declarations of the same function (perhaps from an included header), their types along with the function prototype of the function definition are used to form a composite type. It is this composite type that is used for later calls.

If the declarator includes an identifier list,[139] the types of the parameters shall be declared in a following declaration list. 1834

Commentary

identifier list 1827
declare at least one declarator

This requirement on the program (also covered by a constraint) associates each identifier in the identifier list with the type of the corresponding identifier declared in the declaration list.

C++

The identifier list form of function definition is not supported in C++.

1835 In either case, the type of each parameter is adjusted as described in 6.7.5.3 for a parameter type list;

parameter type adjusted

Commentary

The ordering of this and the following C sentence is important. Both array and function types are converted to pointers to the appropriate respective types before the requirement on being an object type applies.

1598 array type adjust to pointer to
1600 function type adjust to pointer to

1836 the resulting type shall be an object type.

Commentary

This sentence clarifies the relative order in which adjustment of parameter types and checking for an object type occurs. Requirements in other parts of the standard specify that a parameter (which is an object with no linkage) have a complete type by the end of its declarator; so declaring a parameter to have an incomplete type would be a constraint violation.

71 parameter
434 parameter linkage
1361 object type complete by end

C++

The only difference, in parameter types, between a function declaration and a function definition specified by the C++ Standard is:

> *The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type.*

8.3.5p6

1837 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

Commentary

The behavior is undefined if there are either fewer arguments or more arguments, passed in a call, than parameters in the function definition. This C sentence is not specific to the form of function definition (i.e., it could be old-style or prototype) and it is not necessary to access any of the parameters within the function definition. Simply calling the function is sufficient to cause the undefined behavior.

Developers did define functions accepting variable numbers of arguments prior to the availability of prototypes and the ellipsis notation (first specified in the C90 Standard). One technique used to access the additional argument values was to take the address of the last declared parameter and perform pointer arithmetic on it (knowledge of stack layout, such as order of arguments pushed and their alignment, was required to make it point at where the additional arguments were placed).

C++

This C situation cannot occur in C++ because it relies on the old-style of function declaration, which is not supported in C++.

Common Implementations

Because of the unknown, assumed to be significant, amount of existing code that passes variable number of arguments to functions defined using the old-style notation, implementations invariably play safe and do not use any special argument passing conventions (i.e., they are invariably passed on the stack) for such definitions. In the case of function defined using prototype notation, implementations invariably assume that the number of arguments passed equals the number of parameters defined. If these numbers differ the results can be completely unpredictable.

Coding Guidelines

This usage may occur in existing code. The cost/benefit issues involved in modifying existing code are discussed elsewhere. If the guideline recommendation specifying the use of prototypes is followed the usage described by this C sentence will not occur in newly written code.

1810 external declaration syntax
1810.1 function declaration use prototype

parameter automatic storage duration

Each parameter has automatic storage duration. 1838

Commentary

parameter 71
parameter 434 linkage
parameter 1593 storage-class
function declaration list 1829 storage-class specifier
automatic 457 storage duration
parameter 1839 scope begins at

This specification can be deduced from other wording in the standard, i.e., a parameter is an object, whose identifier has no linkage and its declaration may not include the storage-class specifier **static**, therefore it has automatic storage duration.

Coding Guidelines

Some coding guideline documents recommend that function parameters (which are defined to have block scope) be treated as being different from other block scope objects. The recommended differences in treatment arise from conceptual ideas about what parameters represent. For instance, considering them as input only values (which implies that they should not be modified in the function body). The rationale for this view of parameters is often derived from other languages where a pass by address method of argument passing is supported (in this case modifications of a parameters value will affect the value of the corresponding object passed as the argument). There is no obvious benefit in having such a guideline recommendation in C.

parameter scope begins at

Its identifier is an lvalue, which is in effect declared at the head of the compound statement that constitutes the function body (and therefore cannot be redeclared in the function body except in an enclosed block). 1839

Commentary

With the introduction of support for variable length arrays in C99, this description (which was in a footnote in the C90 Standard), is no longer completely accurate.

```
1  void f(int p_length, double p_1[p_length][p_length])
2  { /* ... */ }
```

C++

The C++ Standard does not explicitly specify the fact that this identifier is an lvalue. However, it can be deduced from clauses 3.10p1 and 3.10p2.

3.3.2p2 *The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. If the function has a function try-block the potential scope of a parameter ends at the end of the last associated handler, else it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function try-block.*

The layout of the storage for parameters is unspecified. 1840

Commentary

In the past there existed a relatively common practice of accessing the values of different parameters by calculating their addresses using pointer arithmetic (this usage causes undefined behavior because pointer arithmetic is only guaranteed to deliver defined results while it remains within the same pointed-to object, or one past the end of the same object). This specification of behavior (there is no equivalent wording for other objects) is intended to explicitly point out to developers that the standard does not guarantee any particular storage layout for parameters.

pointer 1170 arithmetic defined if same object
storage 1354 layout

C++

The C++ Standard does not explicitly specify any storage layout behavior for parameters.

Other Languages

Those languages (e.g., Ada and CHILL) that do provide a mechanism for specifying how objects are laid out in storage do not usually provide support for specifying the relative layout of parameters. BCPL explicitly specifies that the parameters form a contiguous array. However, it is implementation-defined whether the first parameter can be accessed using the lowest or highest subscript.

Common Implementations

Most implementations do the obvious and parameters are laid out in a contiguous area of storage, with addresses either allocated high to low, or low to high. However, some implementations speed up parameter passing by using registers and may not even allocate storage if it is not needed. The alignment of storage used for parameters can be influenced by factors that do not affect object storage layout in other contexts. For instance, the processor may support instructions that manipulate the stack in fixed-sized units (e.g., the natural size of type **int**) and the choice of alignment used for various types is driven by the requirements of these instructions (which do not affect alignment choice in other contexts). Storage allocation issues for parameters are also discussed elsewhere.

39 alignment
1354 storage layout
449 stack

Coding Guidelines

Making use of storage layout information requires that other unspecified or undefined behavior be used, for instance incrementing pointer values so that they no longer point within the original object. The guideline recommendations dealing with making use of representation information is applicable.

1171 pointer arithmetic undefined
569.1 representation information using

1841 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment.

function entry parameter type evaluated

Commentary

This requirement, on implementations, creates a dependency between the evaluation of an argument whose corresponding parameter is used in the evaluation of the size expression of a parameter having a variably modified type and the evaluation of the size expression of that variably modified parameter. For instance, the required evaluation order in the following call to `f`:

```
1  extern int glob_1,
2             glob_2[10][2];
3
4  void f(int n, int a[n][2])
5  { /* ... */ }
6
7  void g(void)
8  {
9  f(glob_1, glob_2);
10 }
```

is to assign the argument `glob_1`, in the call to `f`, to the parameter `n`, evaluate the type of `a`, and then assign the argument `glob_2` to the parameter `a`.

The order of evaluation of the size expressions of parameters having variably modified types is unspecified. For instance, a strictly conforming program has to make worst-case assumptions about the number of elements required in the arrays passed as arguments to the functions `f_1` or `f_2` in the follow example:

```
1  extern int glob;
2
3  int g(void)
4  {
5  return glob++;
6  }
7
8  /*
9   * The order of evaluation of the following variably modified
```

```
 * parameters is not defined, but there is a sequence point
 * between the two modifications of glob.
 */
void f_1(int a_1[const g()], int a_2[const g()]])
{ /* ... */ }

/*
 * The evaluation of the following variably modified parameters causes
 * undefined behavior, because the same object is modifies more than
 * once between sequence points (a full declarator is a sequence point.
 */
void f_2(int a_1[const glob++], int a_2[const glob++]])
{ /* ... */ }
```

If a function prototype is visible at the point of call the values of the arguments will already have been converted to the types of the parameters, before being passed. There is no actual assignment performed on function entry.

default argument promotions 1009

If the only visible declaration is an old style function declaration the arguments will be subject to the default argument promotions, which means they may not have the same type as the parameter. When the type of the parameter is not compatible with its promoted type it is necessary for the value of the argument to be converted, as if by assignment, to the type of the parameter.

```
#include <stdio.h>

int f(p)
unsigned char p;
{
/* Implementations acts as if argument passed is assigned to p here. */
printf("argument passed = %d\n", p);
}

int main(void)
{
/* Output from first three calls should be the same as from the second three. */
f(0x42);                f(0x0142);                f(0x2142);
f((unsigned char)0x42); f((unsigned char)0x0142); f((unsigned char)0x2142);
}
```

function call preparing for 1004

The standard specifies elsewhere that each parameter is assigned the value of the corresponding argument.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Common Implementations

Some implementations can optimize away the need to perform an assignment because of the way that they load values from storage, into registers. For instance, a load byte instruction may clear all of the other bits in the register as well as loading a byte into it, removing the need to convert the value passed as an argument (any other, more significant, bits are always ignored).

```
#include <stdio.h>

int f(p)
unsigned char p;
{
/*
 * Accessing the storage allocated to p plus some bytes immediately after it may
```

```
 * access the value of the argument actually passed (provided the implementation
 * does not perform the implicit assignment).  However, such an access also
 * makes use of undefined behavior, so in theory any result could be returned.
 */
printf("parameter value = %d, argument passed may have been = %d\n",
                           p,                              *(int *)&p);
}
```

If the argument value is not representable in the parameter type the behavior of most implementations is the same as that for the same situation in an assignment.

Coding Guidelines

Support for variably modified types is new in C99 and the issue of side effects in their evaluation is one of developer education as well as potentially being coding guideline related. Variable length array declarations can generate side effects, a known problem area. The coding guideline issues for full declarators are discussed elsewhere.

1711 object initializer evaluated when

1842 (Array expressions and function designators as arguments were converted to pointers before the call.)

Commentary

These conversions mirror those that occur for the parameter types.

729 array converted to pointer
732 function designator converted to type
1835 parameter type adjusted

1843 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

Commentary

That is, after all the parameters have been assigned the value of the corresponding arguments.

The compound body associated with a function definition is treated the same way as a compound body within a nested inner block.

This C statement assumes that the execution environment has sufficient resources to allocate storage for the objects defined in the called function. The C Standard does not provide any mechanism to check whether there are sufficient resources available to call and execute the designated function. Neither does it say anything about what might happen if there are insufficient resources to execute the function definition.

C90

The C90 Standard does not explicitly specify this behavior.

C++

The C++ Standard does not explicitly specify this behavior.

1844 If the `}` that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

function termination reaching }

Commentary

This case deals with an implicit **`return`** statement without an expression (omitting an expression from an explicit **`return`** statement in a function returning an object type is a constraint violation). There is a body of existing code that omits an explicit **`return`** statement because it is known that the caller does not access a returned value. Rather than specifying this case as a constraint violation, requiring translators to issue a diagnostic, rending a (allegedly large) body of existing code non-conforming the Committee specified undefined behavior (this exact wording also appears in the C90 Standard).

1800 return without expression

C++

Flowing off the end of a function is equivalent to a **return** *with no value; this results in undefined behavior in a value-returning function.*

The C++ Standard does not require that the caller use the value returned for the behavior to be undefined; this behavior can be deduced without any knowledge of the caller.

```
1  int f(int p_1)
2  {
3  if (p_1 > 10)
4     return 2;
5  }
6
7  void g(void)
8  {
9  int loc = f(11);
10
11 f(2); /* does not change the conformance status of the program */
12       // undefined behavior
13 }
```

Other Languages

This behavior is common to many programming languages.

Common Implementations

The usual behavior is for the value of the function call to be whatever happens to be held in the storage location used to hold the return value, when the terminating **}** is reached.

Coding Guidelines

The **}** that terminates a function is reached for one of two reasons:

guidelines 0
not faults

1. it was not intended to occur and the author of the function body has made a mistake. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults,
2. it is intended to occur in certain situations. This intended usage creates a dependency between the calls that do not access the value returned and the control flow in the function definition that results in no defined value being returned in some situations.

Adhering to a guideline recommending that the terminating **}** in a function not be reached may require adding a **return** statement, with an associated expression. Adding such a statement raises various questions, including the following:

- What value should be returned by this statement? On the basis that the statement is never intended to be executed any value will suffice. A case can be made for zero in that this value is more likely to be within the bounds of expected return values and may not have any significant affect on subsequent execution, and a case can be made for returning a value that is likely to have a noticeable affect on subsequent program execution. Returning a fixed value is at least more likely to ensure consistent behavior.
- How will subsequent readers of the function source treat this **return** statement? Without additional information they are likely to assume the value (e.g., a comment or giving it a symbolic name such as `DUMMY_VALUE`) was intended to be returned. The presence of what is essentially a **return** statement has possible costs as well as possible benefits.

Without any obvious cost/benefit for using a **return** statement no guideline recommendation is made here. However, guidelines having other aims (e.g., defensive programming) may recommend the presence of such a statement.

Usage

In the translated source of this book's benchmark programs 0.7% of function definitions contained both `return;` (or the flow of control reached the terminating **}**) and `return expr;`.

1845 EXAMPLE 1 In the following:

```
extern int max(int a, int b)
{
        return a > b ? a : b;
}
```

extern is the storage-class specifier and **int** is the type specifier; **max(int a, int b)** is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
        return a > b ? a : b;
}
```

Here **int a, b;** is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

Commentary

These two forms of parameter declaration are similar because the type of a and b is compatible with its promoted types

C++

The second definition of max uses a form of function definition that is not supported in C++.

1846 139) See "future language directions" (6.11.7). footnote 139

1847 EXAMPLE 2 To pass one function to another, one might say

```
int f(void);
/* ... */
g(f);
```

Then the definition of **g** might read

```
void g(int (*funcp)(void))
{
        /* ... */
        (*funcp)(); /* or funcp()  ...  */
}
```

or, equivalently,

```
void g(int func(void))
{
        /* ... */
        func(); /* or (*func)()  ...  */
}
```

Other Languages

The second declaration of g is the form often used by other languages.

Coding Guidelines

While the second form of declaration of g may appear more intuitive to many developers, this may be because they have relatively little experience using function pointers. The first declaration of g uses the form needed to declare an object as a pointer to a function type and developers that regularly use pointer to function types will be practiced in reading it.

6.9.2 External object definitions

Semantics

---

object external definition

If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier. 1848

Commentary

external definition 1817

An external definition is an external declaration that is also a definition. The presence of an initializer turns a declaration into a definition, irrespective of its linkage. The definition of an object causes storage to be reserved for it.

object reserve storage 1354

C++

The C++ Standard does not define the term *external definition*. The object described above is simply called a *definition* in C++.

Other Languages

The conditions under which a declaration is also a definition of an object vary between languages, depending on the model of separate translation they use.

translation unit syntax 1810

Common Implementations

The base document did not support the use of initializers on declarations that included the **extern** storage-class specifier.

Coding Guidelines

Common usage is for developers to use the term *definition*, rather than *external definition*, to refer to such declarations. There does not appear to be a worthwhile benefit in attempting to change this common usage.

---

tentative definition

A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. 1849

Commentary

This defines the term *tentative definition* (which is only used in this, 6.9.2, subclause). Tentative definitions are a halfway house. They indicate that a declaration might be a definition, but the door is left open for a later declaration, in the same translation unit to be the actual definition or simply another tentative definition.

The concept of tentative definition was needed, in C90, because of existing code that contained what might otherwise be considered to be multiple definitions, in the same translation unit, of the same object. Defining and using this concept allowed existing code, containing these apparent multiple definitions of the same object, in the same translation unit (an external definition of the same in more than one translation unit is a constraint violation) to be conforming.

definition one external 1812

With two exceptions all external object declarations are tentative definitions; (1) a declaration that contains an initializer is a definition, and (2) a declaration that includes the storage-class specifier **extern** is not a definition.

C++

The C++ Standard does not define the term *tentative definition*. Neither does it define a term with a similar meaning. A file scope object declaration that does not include an explicit storage-class specifier is treated, in C++, as a definition, not a tentative definition.

A translation unit containing more than one tentative definition (in C terms) will cause a C++ translator to issue a diagnostic.

```
int glob;
int glob; /* does not change the conformance status of program */
          // ill-formed program
```

Coding Guidelines

The term *tentative definition* is not generally used by developers and tends only to be heard in technical discussion by translator writers and members of the C committee. There does not appear to be a worthwhile benefit in educating developers about this term and the associated concepts. Their current misconceptions (e.g., declarations that include the storage-class specifier **`static`** become definitions at the point of declaration) appear to be harmless.

Multiple *`external-declaration`*'s of the same object are redundant (this general issue is discussed elsewhere), but otherwise harmless (a modification of the type of one of them will result in a diagnostic being issued unless all of the declarations have compatible type, i.e., they are similarly modified).

190 redundant code

1850 If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

object definition implicit

Commentary

This specification requires implementations to create a definition of an object if the translation unit does not contain one (i.e., there is no declaration of the object that includes an initializer). For an object having an incomplete array type the effect of this specification is to complete the type and define an array having a single element. In the case of objects having an incomplete structure or union type the size of the object is needed to define it, which in turn requires a completed type. Thus, the behavior is undefined if an external object definition has an incomplete structure or union type at the end of a translation unit.

1848 object external definition
1853 EXAMPLE tentative array definition
1354 object reserve storage
1465 footnote 109

C++

The C++ Standard does not permit more than one definition in the same translation unit (3.2p1) and so does not need to specify this behavior.

Coding Guidelines

It is common practice to omit the initializer in the declaration of an object. Developers invariably assume that an object declaration that omits a storage-class specifier is a definition (which does eventually become). The fact that an object might not be defined at its point of declaration is purely a technicality.

1851 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

object type for internal linkage

Commentary

This requirement applies at the point of declaration, not at the end of the translation unit. The affect of this difference is illustrated by the following example:

```
static char a[]; /* Internal linkage, undefined behavior.           */
        char b[]; /* External linkage, equivalent to char b[] = {0}; */
```

One consequence of this requirement is that implementations can allocate storage for objects having internal linkage as soon as the declaration has been processed, during translation.

In the case of objects having external linkage the behavior is not undefined if the object has an incomplete array type (see previous C sentence). For objects having no linkage it is a constraint violation if the type of the object is not completed by the end of the declarator.

1361 object type complete by end

Common Implementations

Some implementations (e.g., gcc) support the declaration of objects having a tentative definition and internal linkage, using an incomplete type (which is completed later in the same translation unit).

EXAMPLE linkage

EXAMPLE 1 1852

```
int i1 = 1;         // definition, external linkage
static int i2 = 2;  // definition, internal linkage
extern int i3 = 3;  // definition, external linkage
int i4;             // tentative definition, external linkage
static int i5;      // tentative definition, internal linkage

int i1;             // valid tentative definition, refers to previous
int i2;             // 6.2.2 renders undefined, linkage disagreement
int i3;             // valid tentative definition, refers to previous
int i4;             // valid tentative definition, refers to previous
int i5;             // 6.2.2 renders undefined, linkage disagreement

extern int i1;      // refers to previous, whose linkage is external
extern int i2;      // refers to previous, whose linkage is internal
extern int i3;      // refers to previous, whose linkage is external
extern int i4;      // refers to previous, whose linkage is external
extern int i5;      // refers to previous, whose linkage is internal
```

Commentary

```
1  extern int i1;      // external linkage
2  extern int i2;      // external linkage
3  extern int i3;      // external linkage
4  extern int i4;      // external linkage
5  extern int i5;      // external linkage
6
7  int i1;             // external linkage
8
9  int i1 = 1;         // definition, external linkage
10 static int i2 = 2;  // internal linkage: undefined behavior -> external linkage on previous declaration
11 extern int i3 = 3;  // definition, external linkage
12 int i4;             // external linkage
13 static int i5;      // internal linkage: undefined behavior -> external linkage on previous declaration
```

C++

The tentative definitions are all definitions with external linkage in C++.

EXAMPLE tentative array definition

EXAMPLE 2 If at the end of the translation unit containing 1853

```
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

Commentary

object definition implicit 1850

The implicit initialization (equal to 0), at the end of a translation unit, of the tentative definition of an object describes an effect. In the case of an object declared to be an array of unknown size, the initializer is treated as specifying a single element.

C90

This example was added to the C90 Standard by the response to DR #011.

C++

This example is ill-formed C++.

Coding Guidelines

This usage might be considered suspicious in that declaring an object to have an array type containing a single element is unusual and if it was known that only one element was needed why wasn't this information specified in the declaration. If the usage was unintended it is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. Any intended usage is likely to be rare and thus a guideline recommendation (if shown to be cost effective technically) is not cost effective.

0 guidelines not faults

6.10 Preprocessing directives

preprocessor directives syntax

1854

```
preprocessing-file:
                group_opt
group:
                group-part
                group group-part
group-part:
                if-section
                control-line
                text-line
                # non-directive
if-section:
                if-group elif-groups_opt else-group_opt endif-line
if-group:
                # if     constant-expression new-line group_opt
                # ifdef  identifier new-line group_opt
                # ifndef identifier new-line group_opt
elif-groups:
                elif-group
                elif-groups elif-group
elif-group:
                # elif   constant-expression new-line group_opt
else-group:
                # else   new-line group_opt
endif-line:
                # endif  new-line
control-line:
                # include pp-tokens new-line
                # define  identifier replacement-list new-line
                # define  identifier lparen identifier-list_opt )
                                              replacement-list new-line
                # define  identifier lparen ... ) replacement-list new-line
                # define  identifier lparen identifier-list , ... )
                                              replacement-list new-line
                # undef   identifier new-line
                # line    pp-tokens new-line
                # error   pp-tokens_opt new-line
                # pragma  pp-tokens_opt new-line
                #         new-line
text-line:
                pp-tokens_opt new-line
non-directive:
```

```
                pp-tokens new-line

lparen:
                a ( character not immediately preceded by white-space
replacement-list:
                pp-tokens_opt
pp-tokens:
                preprocessing-token
                pp-tokens preprocessing-token
new-line:
                the new-line character
```

Commentary

It can be seen from the grammar that most terminals do not match any syntax rules in the other parts of the language. The C language essentially contains two independent languages within it. The preprocessor and what might be called the C language proper.

A *text-line* is the sequence of *pp-tokens* that are scanned for macros to substitute and subsequently processed by the C language syntax.

C90

Support for the following syntax is new in C99:

```
group-part:
        # non-directive
control-line:
        # define  identifier lparen ... ) replacement-list new-line
        # define  identifier lparen identifier-list , ... )
                                                replacement-list new-line
```

The left hand side terms *text-line* and *non-directive* were introduced in C99. Their right-hand side occurred directly in the right-hand side of *group-part* in the C90 Standard.

```
text-line:
        pp-tokens_opt new-line
non-directive:
        pp-tokens new-line
```

The definition of *lparen* in C90 was:

```
lparen:
        the left-parentheses character without preceding white-space
```

C++

The C++ Standard specifies the same syntax as the C90 Standard (16p1).

Other Languages

While most assembly languages support some form of preprocessor, few high-level languages include one (Lisp, PL/1 do).[170] A conditional compilation mechanism was recently added to the Fortran Standard[652] (it is specifically aimed at conditional compilation and does not offer macro processing functionality).

While many versions of Unix include a preprocessor that is independent of the C preprocessor (i.e., `m4`[1392]) it has never achieved wide usage (this reason was given for its exclusion from the POSIX Standard). A few other stand-alone preprocessors (often called *macro processors*) have also been written (e.g., ML/I[169]).

Some non-C preprocessors use the character **$** or **%** rather than **#**.

Common Implementations

Because of the relative simplicity of the preprocessor grammar most implementations do not use a table-driven parser approach, but rather process the directives on an ad hoc basis. Some preprocessors operate at a higher level than individual tokens e.g., taking syntax into account.[507, 1468]

The base document did not support the preprocessing directive **#elif** or **defined**. Support for this functionality was added during the early evolution of C.[1180] Also early implementations supported trailing tokens on the same line as preprocessing directives `#endif MACRO_NAME` and some implementations[1340] continue to provide an option to support this functionality. Some early translators did not permit white-space between **#** and **define** Some also required the **#** to be the first character on the line `gcc` supported `...` in macro definitions prior to C99. The syntax used was to give the varying arguments a (developer chosen) name, for instance in:

```
#define debug(file, format, args...) fprintf(file, format, args)
```

following the parameter `args` with `...` specifies that it takes a variable number of arguments. These variable arguments are substituted wherever the named parameter appears in the replacement list (just like other parameters).

The Unix System V C compiler[1392] (plus `gcc` and others) supports the creation and use of what it calls *assertions*. For instance, (the **#** before an identifier signaling its status as a predicate):

```
/* #assert predicate(token-sequence) */

#assert derek (is_male) /* Associate the answer is_male with the predicate derek. */

/* #if #predicate(token-sequence) */

#if #derek(is_female) /* Evaluates to false. */
#endif
/*
 * Implementations supporting these kinds of assertions usually
 * define the predicates system (with answers such as svr4, mach, posix,
 * hpux, etc.), and cpu (with answers such as i386, sparc, m68k, etc.).
 */
#if #cpu(sparc)
/* ... */
#elif #cpu(i386)
/* ... */
#endif
```

The directive:

```
#ident "version X.Y"
```

is used by a number of translators to include version control information in the source code that is also written out to the generated object file. A common technique is to use a sequence of characters, within the string literal, that are recognized by a source code control system (and which can be updated when a new version is checked in)

The matching directives **#version/#endversion** are available in a Bell-Labs translator. It is claimed[63] that developers were 40% more productive when using a version sensitive editor on a multi million line project having many versions (developed over two decades by more than 5,000 developers).

The Plan 9 C compiler intentionally lacks support for the **#if** directive.[1087]

Coding Guidelines

While lines containing non-preprocessing directives are often visually indented (see Figure 1854.1) developers do not generally indent preprocessing directives (see Figure 1854.1). One possible reason for this difference

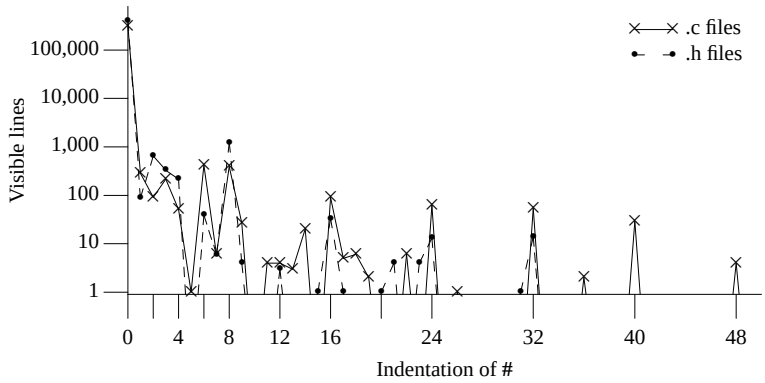


Figure 1854.1: Number of lines containing a preprocessing directive starting at a given indentation from the start of the line (i.e., amount of white space before the first **#** on a line, with the tab character treated as eight space characters). Based on the visible form of `.c` and `.h` files.

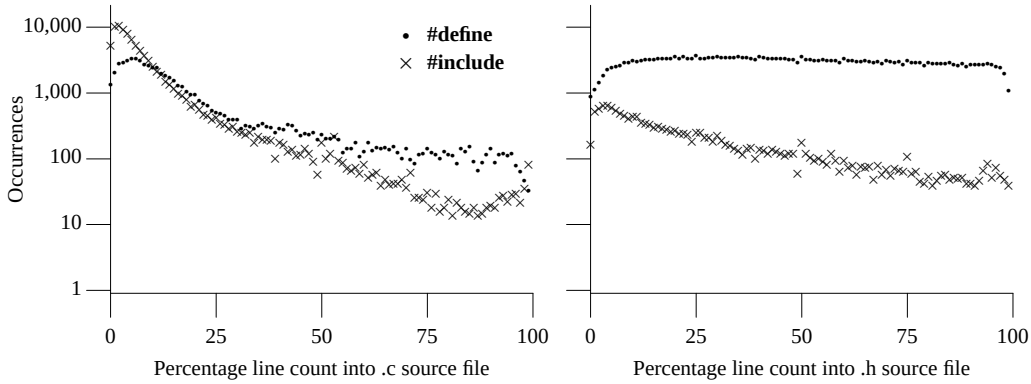


Figure 1854.2: Number of **`#include`** and **`#define`** directives appearing at a relative location (i.e., 100*line_number/lines_in_file) in the source. Based on the visible form of `.c` and `.h` files.

is that indentation of preprocessing directives may not provide a visual edge for readers to run their eyes along (e.g., the visible source between conditional inclusion directives may be statements from the C language whose indentation is driven by factors unrelated to preprocessing; that is there are two indentation schemes operating over the same section of source).

In over 90% of preprocessing directives the **#** character appears at the start of the line. Given that preprocessing directives are usually scattered throughout the source this indentation strategy allows them to generally standout from the surrounding statements.

In general both the **`#include`** and **`#define`** directives appear at the start of a source file (see Figure 1854.2). By their nature the **`#if`** directives are localized at the point when a conditional decision needs to be made.

Usage

A study by Ernst, Badros, and Notkin[396] provides one of the few empirical studies of C preprocessor use.

Table 1854.1: Occurrence of preprocessor directive names and preprocessor operators (as a percentage of all directive names and operators). Based on the visible form of the `.c` and `.h` files.

Directive Name	.c file	.h file	Directive Name	.c file	.h file
`#define`	19.9	75.0	`#if`	6.2	1.5
`#endif`	19.9	7.2	`##`	0.3	0.9
`#include`	28.6	4.1	`#elif`	0.2	0.2
`#ifndef`	2.4	3.2	`#pragma`	0.1	0.1
`#ifdef`	11.3	2.5	`#error`	0.2	0.1
`#else`	4.8	1.7	`#`	0.0	0.1
defined	3.6	1.7	`#line`	1.4	0.0
`#undef`	1.0	1.6			

Description

1855 A ~~preprocessing directive~~ *preprocessingdirective* consists of a sequence of preprocessing tokens that ~~begins with~~ satisfies the following constraints:

preprocessing directive consists of

Commentary

This defines the term *preprocessing directive*. Preprocessor directives are line oriented (the second half of the above sentence requires directive to start on a line and finish on the same line).

The wording was changed and the sentence split up by the response to DR #303.

1856 The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character ~~,~~ .

Commentary

It is only after preprocessing that the C language becomes is free format. Line splicing occurs prior to translation phase 4 and was created to enable preprocessor directives to span more than one physical source line. Because preprocessing directives are identified at the start of translation phase 4 translators do not treat the output from macro expansion (which occurs during translation phase 4) as possible preprocessing directives (or handle other unusual token sequences as preprocessing directive).

The wording was changed, and this sentence formed, by the response to DR #303.

118 line splicing
129 translation phase 4
1973 expanded token sequence not treated as a directive
1868 EXAMPLE EMPTY #

C90

The C90 Standard did not contain the words "(at the start of translation phase 4)", which were added by the response to DR #144.

C++

Like C90, the C++ Standard does not specify the start of translation phase 4.

1857 ~~and is ended by the next~~ The last token in the sequence is the first new-line character that follows the first token in the sequence. [140)]

Commentary

The wording was changed, and this sentence formed, by the response to DR #303.

1858 A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

preprocessing directive ended by

Commentary

In the following:

```
#define M_1(a, b) ((a) + (b))
#if M_1(1,              /* Syntax violation. */
          2) == 3       /* Different logical line unconnected with the previous one. */
#endif

void f(void)
{
int loc = M_1(1,
                2); /* No end-of-line syntax requirements here. */
}
```

the **#if** preprocessing directive visually spans more than one line. The fact that the macro invocation M_1 is split across two physical lines does not alter the requirements on this preprocessing directive (and a translator required to generate a diagnostic). However, the following example is a rather unusual case:

macro invocation) terminates it 1924

```
#define M_1(a, b) ((a) + (b))
#if M_1(1, /* Comment about the first argument.
              Comment about the second argument */ 2) == 3
#endif
```

in that the comment is replaced by one space character and the new-line character *seen* by translation phase 4 is the one that occurs after the preprocessing token 3 (i.e., the code is conforming).

translation phase 3 124

C90

This wording was not in the C90 Standard, and was added by the response to DR #017.

C++

Like the original C90 Standard, the C++ Standard does not explicitly specify this behavior. However, given that vendors are likely to use a preprocessor that is identical to the one used in their C product (or the one that used to be used, if they nolonger market a C product), it is unlikely that the behaviors seen in practice will be different.

---

footnote 140

140) Thus, preprocessing directives are commonly called “lines”. 1859

Commentary

The common developer usage is for the term *lines* to refer to any sequence of characters terminated by a new-line, i.e., the use is not specific, in the developer community, to preprocessing directives.

Coding Guidelines

Developers are aware of the line-oriented nature of preprocessing directives. However, no terminology (other than *preprocessing directive*, or simply *directive*) is commonly used.

These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 6.10.3.2, for example). 1860

Commentary

All source file lines are processed during translation phase 4, either as directives or as token sequences to examine for possible macro replacement. However, the preprocessing directive *lines* are deleted at the end of translation phase 4 and play no further role in translation.

preprocessing directives deleted 132

A text line shall not begin with a # preprocessing token. 1861

Commentary

This is a requirement on the implementation. Syntactically a *text-line* can start with a # (which is a preprocessing token). This requirement disambiguates the syntax by specifying that such an interpretation shall not be made by the implementation.

1862 A non-directive shall not begin with any of the directive names appearing in the syntax.

Commentary

This is a requirement on the implementation. Syntactically any sequence of *pp-tokens* following a # preprocessing token can be a *non-directive*. This requirement disambiguates the syntax by specifying that such an interpretation (if one exists for the given sequence of preprocessing tokens) shall not be made by the implementation where there is another interpretation.

C90

Explicit support for *non-directive* is new in C99.

C++

Explicit support for *non-directive* is new in C99 and is not discussed in the C++ Standard.

1863 When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Commentary

This C sentence introduces the term *directive name* to describe the preprocessing tokens that occur immediately after the # punctuator (they can be given the status of keywords). 1874 footnote 141

The Committee recognized that the source code contained within a skipped group might still be under development (which is one reason for it being skipped). Given that comments do not nest, it may not be possible, or desirable, to use a comment to prevent translators analyzing the sequence of characters in detail. This relaxation of the syntax allows conditional inclusion directive to be used to bracket a sequence of lines that do not yet form a conforming sequence of *pp-tokens*. 932 footnote 69 1890 directive processing while skipping

```
#if WORKING_ON_IT

#if don't know what condition to put here yet
#endif

#endif
```

The directive name needs to be processed to handle any nested conditional preprocessing directives.

C90

The C90 Standard did not explicitly specify this behavior.

C++

Like C90, the C++ Standard does not explicitly specify this behavior.

Common Implementations

Many implementations attempt to process skipped groups as quickly as possible. By default, many implementors only examine the minimum number of preprocessing tokens necessary to work out where the skipped group ends.

Constraints

1864 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3). white-space within preprocessing directive

Commentary

white-space characters 778

The use of other white-space characters, (i.e., vertical-tab or form-feed) within preprocessing directives could cause the visual appearance of the source code to suggest that directives are terminated by new-line when in fact they are not. That is the source would be visually confusing and such usage has no obvious benefit. These white-space characters may still appear within preprocessing tokens, e.g., in a character string literal.

white-space sequence replaced by one 128

It is implementation-defined whether other white-space characters are replaced by one space character in translation phase 3.

Common Implementations

Some implementations support the appearance of a carriage return character immediately prior to the end of line (when that character is not part of the end of line sequence). This enables them to handle text files written using the MS-DOS end of line conventions (i.e., end of line is represented by the two characters \r\n).

Coding Guidelines

The presence of white-space within a preprocessing directive may or may not be significant. It is sometimes needed (and invariably appears, whether needed or not) to separate the definition of an object-like macro identifier from its body. For instance, in:

macro object-like 1931

```
1  #define A (x)
2  #define B-y
```

without the intervening white-space the first definition would be for a function-like macro taking a parameter x. No white-space is used in the second definition, but the source would probably be easier to process if some were used.

Semantics

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. 1865

Commentary

#pragma directive 1994

translation phase 4 129

Preprocessing directives (e.g., **`#pragma`**) can also affect how other phases of translation behave. These operations are performed during translation phase four.

Coding Guidelines

It is possible to completely change the meaning of some C constructs, compared to the behavior that would be deduced purely from their visual appearance in the source code. Constructs can be added, deleted, and modified. All without any indication that these operations will occur, in the source code (or at least the `.c` files). While various criticisms are made about the behavior of the preprocessor (e.g., it performs no semantic checks on the substitutions it makes) the actual problem lies with the people who use it. The preprocessor might be likened to a sharp knife (some people might say a large hammer), being sharp makes the job of cutting easy but don't complain if sloppy use results in the user being cut (or objects near to the hammer being crushed through poor aiming).

In some cases it is possible to provide guideline recommendations that prevent problems occurring, in other cases complexity is intrinsic to the problem being handled. For instance, the heavy use of **`#if`**/**`#endif`** may be the result of having a successful product that runs in many different environments. While some coding guideline documents recommend that preprocessor usage be minimized, such recommendations are really a design issue and as such are outside the scope of these coding guidelines.

---

preprocessing

These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit. 1866

Commentary

This defines the term *preprocessing* (the term *macro processing* is sometimes used by developers, perhaps because macro substitution is felt to have the largest impact on the visible form of the source). This is also a generally used term within software engineering that refers to any set of operations that are performed on the input prior to what are considered to be the main operations (in C's case the subsequent phases of translation). Any later operations are sometimes referred to as *post processing*. The component of the translator that performs these operations (independent of whether it is a stand alone program or an integrated part of a larger program) is universally known as the *preprocessor*.

The output from preprocessing is a translation unit.

110 translation unit known as

The C preprocessor is sometimes used to provide macro processing and conditional compilation in non-C language contexts (e.g., other languages and text processing tasks). Members of the C committee have at times born this usage in mind when considering how the preprocessor should behave.

Common Implementations

In many implementations preprocessing is carried out by a stand-alone program, usually known as the *preprocessor*.

120 footnote 5

Coding Guidelines

Developers often think of the output of translation stage four as being the output from the preprocessor. In fact the preprocessor is really only translation phase 4, phases 1–3 would need to be performed even if C did not include a preprocessor (the operations these phases perform are performed by languages that do not include a preprocessor).

Many guideline recommendations are driven by how developers visually process source code. The recommendations that involve preprocessor constructs are often made because the processing that occurs (e.g., macro expansion) is not directly visible to readers of the source.

1867 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

tokens in directive not expanded unless

Commentary

Requiring that macro expansion unconditionally occur within preprocessing directives is not necessarily a desirable requirement (it would also be a change to existing translator behavior that could break existing code). Most preprocessing directives have a form in which macro expansion can occur.

1876 #if macros expanded
1905 #include macros expanded
1991 #line macros expanded
1943 footnote 146

Example

```
#define defined 1 +

#if defined(X) /* Not expanded. */
#endif
```

1868 EXAMPLE In:

EXAMPLE EMPTY #

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a **#** at the start of translation phase 4, even though it will do so after the macro **EMPTY** has been replaced.

Commentary

This behavior simplifies the job of tools that scan C source (in that they only need to check the first two preprocessing tokens at the start of a line). However, occurrences of EMPTY in other contexts, within a preprocessing directive, may cause complications.

```
#define EMPTY

#include EMPTY <stdio.h>
#include <stdio.h> EMPTY
```

C90

This example was not in the C90 Standard and was added by the response to DR #144.

C++

This example does not appear in the C++ Standard.

6.10.1 Conditional inclusion

conditional inclusion

Constraints

---

conditional inclusion constant expression

The expression that controls conditional inclusion shall be an integer constant expression except that: it shall not contain a cast; 1869

Commentary

An expression occurring in this context has to be evaluated during translation, hence the need for it to be a constant. The additional constraints are designed to allow the preprocessor to operate independently of the subsequent phases of translation.

Coding Guidelines

source file inclusion 1896

Developers do not commonly refer to this construct using the term *conditional inclusion* (which is suggestive of source file inclusion, i.e., the **#include** directive). Some of the terms that are commonly used by developers include *hash if* and *conditionally compiled*. There does not appear to be a worthwhile benefit in attempting to change existing, developer, use of terminology. However, coding guideline documents still need to make it clear what any terms they use might use, refer to.

---

it shall not contain a cast; 1870

Commentary

footnote 141 1874

This constraint is redundant for the reasons pointed out in footnote 141 (there is no constraint given here against **sizeof** appearing in this context). While it is possible to write a cast that will evaluate to a constant expression under the rules applied to conditional inclusion expressions (e.g., `(int * const)+1` will evaluate to 1), in the majority of cases any cast occurring in this context will result in a syntax violation. The preprocessor is not required to have any knowledge of the representation used for any scalar types by subsequent phases of translation. The type of the operands of the constant expression are all defined to have the same rank.

#if operand type uintmax_* 1880

---

identifiers (including those lexically identical to keywords) are interpreted as described below;[141] 1871

Commentary

#if macros expanded 1876
#if identifier replaced by 0 1878

Identifiers are either subject to macro expansion, treated as an operator (see next C sentence), or replaced by 0.

1872 and it may contain unary operator expressions of the form

#if defined

```
defined identifier
```

or

```
defined ( identifier )
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

Commentary

Semantics in a Constraints clause.

Rationale

The operator **defined** was added to C89 to make possible writing boolean combinations of defined flags with one another and with other inclusion conditions.

The **defined** operator allows more than one test to be made in a single expression. Without this operator it would be necessary to use nested **#ifdef** or **#ifndef** directives, for instance:

```
#ifdef ...
   #ifdef ...
      #ifdef ...
```

Coding Guidelines

The result of the **defined** operator has a boolean role.

476 boolean role
943.1 expression shall be parenthesized

The rationale for using the parenthesized form of the **sizeof** operator is not applicable to the **defined** operator. However, existing practice is to use the parenthesized form (see Table 1872.1). The most common usage of the **defined** operator, in the `.c` files, is equivalent to using the **#ifdef** preprocessing directive.

1884 #ifdef

Example

```
#define X Y

#if defined(X) /* Checks that X is defined, not Y */
#endif
```

Table 1872.1: Occurrence of controlling expressions containing the **defined** operator (as a percentage of all **#if** and **#elif** preprocessing directives). The **#elif** preprocessing directive was followed by the **defined** operator in 66.5% of occurrences of that preprocessing directive— in the .c files (.h 75.5%). Based on the visible form of the .c and .h files.

Preprocessing Directive	%
#if defined **(** identifier **)**	15.7
#if defined **(** identifier **)** **||** defined **(** identifier **)**	5.8
#if defined **(** identifier **)** **&&** defined **(** identifier **)**	2.0
#if **!** defined **(** identifier **)**	1.9
#elif defined **(** identifier **)**	1.9
#if defined **(** identifier **)** **&&** **!** defined **(** identifier **)**	1.3
#if **!** defined **(** identifier **)** **&&** **!** defined **(** identifier **)**	0.9
#if defined **(** identifier **)** **||** defined **(** identifier **)** **||** defined **(** identifier **)**	0.8
#if defined identifier **||** defined identifier	0.5
#if **!** defined **(** identifier **)** **&&** **!** defined **(** identifier **)** **&&** **!** defined **(** identifier **)**	0.3
others	5.3

Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token (6.4). 1873

preprocessing token shall have lexical form 771

Commentary

This requirement duplicates one given elsewhere.

This sentence was added by the response to DR #304.

footnote 141

141) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names— there simply are no keywords, enumeration constants, etc. 1874

preprocessing token converted to token 137

Commentary

Preprocessing tokens are converted to tokens during translation phase 7.

Coding Guidelines

#if identifier replaced by 0 1878

A common mistake made by developers is to attempt to use the **sizeof** operator within a controlling constant expression. Once the **sizeof** is replaced by 0 the resulting expression is very likely to contain a syntax violation (i.e., consideration of a guideline recommendation is unnecessary because the syntax violation will cause a diagnostic to be generated).

Semantics

Preprocessing directives of the forms 1875

**if** *constant-expression new-line* $group_{opt}$
**elif** *constant-expression new-line* $group_{opt}$

check whether the controlling constant expression evaluates to nonzero.

Commentary

Unlike the controlling expression of an **if** statement it is not necessary to enclose the expression in parentheses. Also the **elif** form does not exist in C language. There is no **#switch** to handle a sequence of **#if**/**#else** or **#elif** preprocessor directives (although such a construct has been proposed as a worthwhile extension, the C Committee continues to turn it down).

Rationale **#elif** was added to minimize the stacking of **#endif** directives in multi-way conditionals.

Other Languages

Some languages (e.g., Algol 68 and Ada) also include support for the **elif** keyword.

Common Implementations

Snelting[1261] applied concept analysis to conditional inclusion directives to produce a visualization of the structure and properties of possible program build configurations.

1821 concept analysis

Coding Guidelines

The coding guideline issues that are applicable to the **if** statement are also applicable to conditional inclusion directives.

1739 selection statement syntax

Experience suggests that developers do not always use the **#elif** form in contexts where it could be used. The following are possible reasons for this behavior:

- Developers having a mental model of conditionals that does not include this construct (an equivalent construct is not available within an **if** statement).
- Developers treating **#if**/**#endif** pairs as conceptually independent of any preprocessing directives they may be nested within.
- The result of cut-and-paste operations from other parts of the source.
- Portability. One common reason why source code contains many nested conditional directives is to handle the behavior of many different translators and environments that is has been ported to. Given that historically some translators have not supported the **#elif** directive, of necessity the source may have had any use of this directive removed.

At the time of this writing there is insufficient information available to make a cost/benefit decision on a guideline recommendation and none is given here.

form of representation mixing

The *constant-expression* in a **#if** directive is sometimes evaluated by readers of the source. The expression may contain integer constants represented using a variety of different forms (e.g., decimal, hexadecimal, character constant, etc.).

A study by Gonzalaz and Kolers[505] investigated how subject's performance on an arithmetic problem was affected by the notational system used to represent values. Subjects were shown an equation of the form $p + q = n$ and had to respond true/false as quickly as possible (e.g., does $2 + 3 = 6$?). The numeric values were presented using either Arabic or Roman digits (e.g., $V + 3 = IX$, $4 + II = 6$, etc.). The results were not consistent with a model that translated the numerals into a single representation system before adding and comparing them. Others[1404] have suggested that the difference in performance can be explained by the time taken to encode the different representations.

Mixed notation can occur in C source. For instance, an expression can contain integer constants represented using decimal, hexadecimal, and octal lexical forms. The results of these studies show that there is a cognitive cost associated with mixing different representations in the same expression. There is no obvious guideline recommendation for trading off this cost against the benefit of using different representations (for different expressions).

Usage

The visible form of the `.c` files contained 12,277 (`.h` 4,159) **`#else`** directives.

Table 1875.1: Common **`#if`** preprocessing directive controlling expressions (as a percentage of all **`#if`** directives). Where *`integer-constant`* is an integer constant expression, and *function-call* is an invocation of a function-like macro. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
identifier	26.5
`integer-constant`	20.3
defined (identifier)	16.4
defined (identifier) || defined (identifier)	6.0
identifier == identifier	2.4
identifier > *`integer-constant`*	2.4
identifier >= function-call	2.1
defined (identifier) && defined (identifier)	2.0
! defined (identifier)	2.0
defined (identifier) && ! defined (identifier)	1.3
identifier >= *`integer-constant`*	1.3
identifier != *`integer-constant`*	1.1
identifier < function-call	1.1
! identifier	1.1
others	14.0

Table 1875.2: Common **`#elif`** preprocessing directive controlling expressions (as a percentage of all **`#elif`** directives). Where *`integer-constant`* is an integer constant expression, and *function-call* is a function-like macro. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
defined (identifier)	49.7
identifier == identifier	19.4
defined identifier	6.6
defined (identifier) || defined (identifier)	5.7
identifier	4.7
defined (identifier) && defined (identifier)	2.6
identifier == *`integer-constant`*	1.9
identifier >= function-call	1.2
defined (identifier) || defined (identifier) || defined (identifier)	1.2
identifier >= *`integer-constant`*	1.0
others	6.1

#if macros expanded

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **`defined`** unary operator), just as in normal text. 1876

Commentary

The following quote from DR #258 has had some lines and quotes (from the C Standard) removed.

#258 ***Problem***

Consider the code:

```
#define repeat(x) x && x    // Line 1
#if repeat(defined fred)    // Line 2
```

and the code:

```
#define forget(x) 0         // Line 3
#if forget(defined fred)    // Line 4
```

Does line 2 "generate" a ***defined*** *operator? Is line 4 strictly conforming code, or does the fact that macro expansion "forgets" the* ***defined*** *operator cause a problem ?*

I would guess that the original intention was that any `defined X` *pair in the original source worked correctly. The proposed change would resolve this.*

In addition, given the order of events, it is unsuitable to say that a `defined X` *expression is "evaluated". Rather it should be described as a textual substitution.*

Proposed Committee Response

The standard does not clearly specify what happens in this case, so portable programs should not use these sorts of constructs.

1877 If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined.

Commentary

Some implementations support this form and there is existing code that makes use of it. However, the C committee did not want to require this usage to be supported. The behavior is also undefined if the **defined** operator is the subject of a **#define** directive.

2026 predefined macros not #defined

Common Implementations

Implementations vary in the order in which they process the sequence of preprocessing tokens in the constant expression. Some perform a pass that handles any **defined** operators, while others perform all processing (e.g., including macro replacement) in a single pass.

10 implementation single pass

Coding Guidelines

Intentional use of this behavior is likely to involve a directive that is either successfully translated or causes a diagnostic to be issued. This largely removes one rationale for a guideline recommendation. Occurrences of uses of this behavior (which may involve developers needing to comprehend what is occurring) are considered to be rare. Which removes the other rationale for a guideline recommendation.

Example

```
#if DEF
#define FOO defined
#else
#define FOO 1 +
#endif

#if FOO BAR
#endif

#define PAREN_PLUS ) + 1

#if defined(X PAREN_PLUS
#endif

#define M(a) defined(a)

#if M(X)
#endif
```

#if identifier replaced by 0

After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. 1878

Commentary

Requiring that all identifiers appearing in this context be defined as a macro has the potential for introducing a great deal of complexity. Zero has several useful arithmetic properties that enable it to occur with no, or a canceling, affect. Being able to use it as an implicit initial value, for identifiers that may be defined as macros, reduces the number of possibilities that developers need to consider.

preprocessing token converted to token 137

The conversion of a preprocessing token to a token mirrors that which occurs in translation phase 7. The sequencing rules mean that any remaining identifiers are converted to the pp-number **0** before token conversion occurs.

The wording was changed by the response to DR #305.

C++

In the C++ Standard **true** and **false** are not identifiers (macro names), they are literals:

16.1p4 *. . . , except for* ***true*** *and* ***false****, are replaced with the pp-number 0, . . .*

If the character sequence `true` is not defined as a macro and appears within the constant-expression of a conditional inclusion directive, when preprocessed by a C++ translator this character sequence will be treated as having the value one, not zero.

Coding Guidelines

The analogy might be made between an identifier that has not been defined as a macro and an object that has not been explicitly initialized. However, use of this analogy requires a choice to be made, (1) for static storage duration an implicit value of zero is provided, and (2) while for automatic and allocated storage duration the value of the object is indeterminate. If the usage was unintentional, it is a fault and considered to be outside the scope of these coding guidelines. An intentional usage may cause subsequent readers to spend more time deducing that the affect of the usage is to produce the value zero, than if they had been able to find a definition that explicitly specified a zero value.

guidelines 0 not faults

The simplest way of adhering to a guideline recommending that all identifiers appearing in the controlling expression of a conditional inclusion directive be defined would be to insert the following (replacing `X` by the undefined identifier) on the lines before it:

```
#ifndef X
#define X 0
#endif
```

However, given these potential usage patterns there does not appear to be a worthwhile benefit in a guideline recommendation covering this issue.

Example

In the following the developer may be expecting `M` to be replaced by the body of some previously defined macro. If no such macro exists `M` will be replaced by 0.

```
#if M == 2
#endif
```

Usage

Approximately 15% of all conditional inclusion directives, in the translated form of this book's benchmark programs, contained an identifier that was replaced by 0 (i.e., they contained an identifier that was neither the operand of **defined** or defined as macro names).

1879 The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6, ~~except that all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header **<stdint.h>**~~.

Commentary

The wording was changed by the response to DR #265.

#if operand type uintmax_*

1880 For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header **<stdint.h>**.[142)]

Commentary

The preprocessor has to evaluate constant expressions using one or more integer type representations. Limiting the representations used to integer types having the same rank minimizes the number of dependencies between the preprocessor and the translator (this requirement also creates a dependency between the processor and the contents of the header **<stdint.h>** used by the implementation).

The types `intmax_t` and `uintmax_t` are the signed or unsigned version of the greatest-width integer type supported by the implementation (and have rank at least equal to that of the type **long long**). Any differences in the representation the types `intmax_t` and `uintmax_t` used by the preprocessor and the typedef names defined (if any, through the appropriate **#include**) in a translation unit would mean that an implementation was not conforming (the situation is the same as that between the type of **sizeof** and the `size_t` used from **<stddef.h>**, see DR #017q7).

The wording was changed by the response to DR #265.

C90

The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 6.4 using arithmetic that has at least the ranges specified in 5.2.4.2, except that ***int*** *and* ***unsigned int*** *act as if they have the same representation as, respectively,* ***long*** *and* ***unsigned long****.*

The ranks of the integer types used for the operands of the controlling constant expression differ between C90 and C99 (although in both cases the rank is the largest that an implementation is required to support). Those cases where the value of the operand exceeded the representable range in C90 (invariably resulting in the value wrapping) are likely to generate a very large value in C99.

C++

The C++ Standard specifies the same behavior as C90 (see the C90 subsection above).

Common Implementations

While a number of C90 implementations supported the type **long long** they still performed preprocessor arithmetic using the types **long** and **unsigned long** (as specified by the C90 requirements). Fear of breaking existing source code means that vendors are likely to offer some form of C90 compatibility option that will continue to perform preprocessor arithmetic using the types specified in C90.

Coding Guidelines

There are a number of possible coding guideline issues associated with the value of a constant expression in a **#if** directive, including:

- The value may be different from the value of an identical sequence of tokens in other contexts in the source file (e.g., the right operand of an assignment statement).
- The value may depend on the implementation used (this problem is not preprocessor-specific, the representation used for the operands in an expression can depend on the implementation).

- The specification has changed between C90 and C99.

The problem with any guideline recommendation is that the total cost is likely to be greater than the total benefit (a cost is likely to be incurred in many cases and a benefit obtained in very few cases). For this reason no recommendation is made here. The discussion on suffixed integer constants is also applicable in the context of a conditional inclusion directive.

integer constant 835
type first in list

Example

In the following the developer may assume that unwanted higher bits in the value of `C` will be truncated when shifted left.

```
#define C 0x1100u
#define INT_BITS 32

#define TOP_BYTE (C << (INT_BITS-8))

#if TOP_BYTE == 0
/* ... */
#endif

void f(void)
{
if (TOP_BYTE == 0)
   /* ... */ ;
}
```

This includes interpreting character constants, which may involve converting escape sequences into execution character set members. 1881

#if escape sequences

Commentary

This conversion also occurs in translation phase 5.

translation phase 5 133

Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.[143] 1882

Commentary

The C committee recognized that developers may choose to perform different phases of translation on different hosts. For instance, source files may be preprocessed and then distributed for further translation on other, different, hosts.

Common Implementations

Differences between the numeric values in these two cases is rare (although cases involving Ascii and EBCDIC character sets do occur).

EBCDIC 3

Coding Guidelines

Making use of the numeric value of character constants is making use of representation information, which is covered by a guideline recommendation. However, there are cases where deviations may occur.

representation information 569.1
representation information using 569.1
using

Example

See footnote 141.

footnote 141 1874

Also, whether a single-character character constant may have a negative value is implementation-defined. 1883

basic character set may be negative

Commentary

The guarantee on the value being nonnegative does not apply during preprocessing. For instance, a preprocessing using the EBCDIC character set and acting as if the type **char** was signed. In other contexts the value of a character constant containing a single-character that is not a member of the basic execution character set is implementation-defined.

478 basic character set positive if stored in char object

885 character constant more than one character

Coding Guidelines

The discussion on the possibility of character constants having other implementation-defined values is applicable here.

885 character constant more than one character

1884 Preprocessing directives of the forms

#ifdef #ifndef

```
# ifdef  identifier new-line group_opt
# ifndef identifier new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name.

Commentary

There is no **#elifdef** form (although over half of the uses of the **#elif** directive are followed by a single instance of the **defined** operator— Table 1872.1).

1885 Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

Commentary

The **#ifdef** and **#ifndef** forms are rather like the unary **++** and **--** operators in that they provide a short hand notation for commonly used functionality.

Coding Guidelines

The **#ifdef** forms are the most common form of conditional inclusion directive. Measurements (see Table 1872.1) also show that nearly a third of the uses of the **defined** operator could be replaced by one of these forms. There are advantages (e.g., most common form suggests most practiced form for readers, and ease of visual scanning down the left edge of the source) and disadvantages (e.g., requires more effort to add additional conditions to the single test being made) to using the **#ifdef** forms, instead of the **defined** operator. However, there does not appear to be a worthwhile cost/benefit to recommending one of the possibilities.

1886 142) Thus on an implementation where INT_MAX is 0x7FFF and UINT_MAX is 0xFFFF, the constant 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7.

Commentary

The wording was changed by the response to DR #265.

1887 143) Thus, the constant expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

footnote 143

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

Commentary

This situation could occur, for instance, if the Ascii representation were used during the preprocessing phases and EBCDIC were used during translation phase 5.

133 translation phase 5

1888 Each directive's condition is checked in order.

Commentary

The order is from the lowest line number to the highest line number.

Coding Guidelines

selection statement 1739 syntax

It may be possible to obtain some translation time performance advantage (at least for the original developer) by appropriately ordering the directives. Unlike developer behavior with **if** statements, developers do not usually aim to optimize speed of translation when deciding how to order conditional inclusion directives (experience suggests that developers often simply append new directive to the end of any existing directives).

Recognizing a known pattern in a sequence of directives has several benefits for readers. They can make use of any previous deductions they have made on how to interpret the directives and what they represent, and the usage highlights common dependencies in the source. In the following code fragment more reader effort is required to spot similarities in the sequence that directives are checked than if both sequences of directives had occurred in the same order.

```
1   #ifdef MACHINE_A
2   /* ... */
3   #else
4   #ifdef MACHINE_B
5   /* ... */
6   #endif
7   #endif
8
9   #ifdef MACHINE_B
10  /* ... */
11  #else
12  #ifdef MACHINE_A
13  /* ... */
14  #endif
15  #endif
```

guideline recommendation 0 enforceable

Given the lack of attention from developers on the relative ordering of directives and the benefits of using the same ordering, where possible, a guideline recommendation appears worthwhile. However, a guideline recommendation needs to be automatically enforceable and determining when two sequences of directives have the same affect, during translation, may be infeasible because information that is not contained within the source may be required (e.g., dependencies between macro names that are likely to be defined via translator command line options).

Rev 1888.1

Where possible the visual order of evaluation of expressions within different sequences of nested conditional inclusion directives shall be the same.

If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; 1889

Commentary

if statement 1744 operand compare against 0

A parallel can be drawn with the behavior of **if** statements, in that if their controlling expression evaluates to zero, during program execution, any statements in the associated block are skipped.

---

directive processing while skipping

directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; 1890

Commentary

The preprocessor operates on a representation of the source written by the developer, not translated machine code. As such it needs to perform some processing on its input to be able to deduce when to stop skipping.

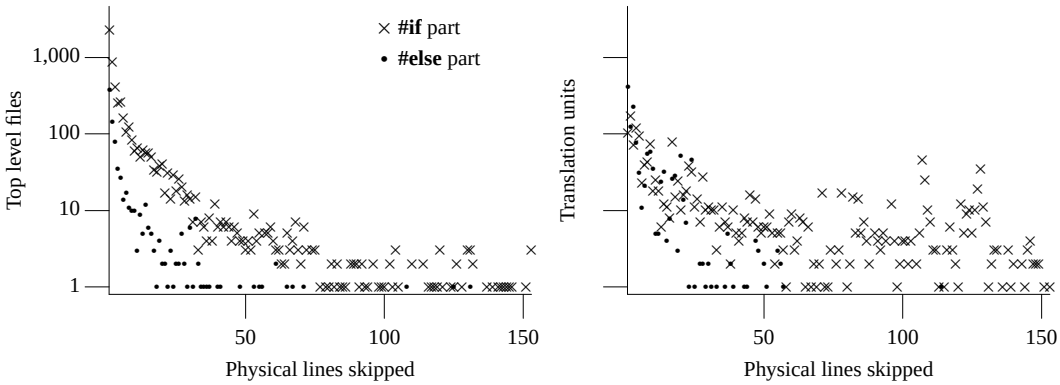


Figure 1889.1: Number of top-level source files (i.e., the contents of any included files are not counted) and (right) complete translation units (including the contents of any files **#include**d more than once) having a given number of lines skipped during translation of this book's benchmark programs.

Directives need to be processed to keep track of the level of nesting of conditionals and translation phases 1–3 still need to be performed (line splicing could affect what is or is not the start of a line) and characters within a comment must not be treated as directives.

116 translation phase 1

The intent of only requiring a minimum of directive processing, while skipping, is to enable partially written source code to be skipped and to allow preprocessors to optimize their performance in this special case, speeding up the rate at which the input is processed.

Example

```
#if 1
extern int ei;

#elif " an unmatched quote character, undefined behavior

extern int foo_bar;
#endif

#if 0
printf("\
#endif \n");

#endif

#if 0
/*
#endif
*/
#endif
```

1891 the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group.

Commentary

There is no requirement that any directive be properly formed, according to the preprocessor syntax. However, preprocessing tokens still need to be created, before they are ignored (as part of translation phase 3).

1854 preprocessor directives syntax
124 translation phase 3

Example

In the following the **#define** directive is not well formed. But because this group is being skipped the translator is required to ignore this fact.

```
#if 0
#define M(e
#endif
```

Only the first group whose control condition evaluates to true (nonzero) is processed. 1892

Commentary

This group is processed exactly as-if it appeared in the source outside of any group.

If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; 1893

Commentary

A semantic rule to associate **#else** with the lexically nearest preceding **#if** (or similar form) directive, like the one given for **if** statements, is not needed because conditional inclusion is terminated by a **#endif** directive.

else 1747 binds to nearest if

Like the matching **#if** (or similar form) directive case, all preprocessing tokens in the group are treated as if they appeared outside of any conditional inclusion directive. Processing continues until the first **#endif** is encountered (which must match the opening directive).

Coding Guidelines

The arguments made for **if** statements always containing an **else** arm might be thought to also apply to conditional inclusion. However, the presence of a matching **#endif** directive reduces the likelihood that readers will confuse which preprocessing directive any **#else** associates with (although other issues, such as lack of indentation or a large number of source lines between directives can make it difficult to visually associate matching directives).

else 1745

---

lacking a **#else** directive, all the groups until the **#endif** are skipped.[144)] 1894

Commentary

The affect of this specification mimics the behavior of **if** statements.

else 1747 binds to nearest if

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.18.1.5). 1895

6.10.2 Source file inclusion

Constraints

---

source file inclusion

A **#include** directive shall identify a header or source file that can be processed by the implementation. 1896

Commentary

There is no requirement that a header be represented using a source file. It could be represented using prebuilt information within the translator that is enabled only when the appropriate **#include** directive is encountered during preprocessing (but not in a group that is skipped). Also there is no requirement that the spelling of the header in the C source file be represented by a source file of the same spelling. The C Standard has no explicit knowledge of file systems and is silent on the issue of directory structures. Minimum required limits on the implementation processing of a header name are specified elsewhere.

footnote 2018 153

#include 1909 mapping to host file

Failure to locate a header or source file that can be processed by the implementation (e.g., a file of the specified name does not exist, at least along the places searched) is a constraint violation.

Other Languages

Most languages do not specify a **#include** mechanism, although many of their implementations provide one. The approach commonly used by C implementations is popular, but not universal. Some languages explicitly state that a **#include** directive denotes a file of the given name in the translators host environment.

Common Implementations

For most implementations the header name maps to a file name of the same spelling. It is quite common for the translation environment to ignore the case of alphabetic letters (e.g., MS-DOS and early versions of Microsoft Windows), or to limit the number of significant characters in the file name denoted by a header name (the remaining characters being ignored). Use of the / character in specifying a full path to a file is sufficiently common usage that even host environments where this character is not normally associated with a directory separator support such usage in header names (many Microsoft windows translators support this character, as well as the \ character, as a directory separator).

In the majority of implementations **#include** directives specify files containing source in text form. However, some implementations support what are known as *precompiled headers*.

121 source file representation
121 header precompiled

It is not uncommon (over 10% of **#include**s in Figure 1896.1) for the same header to be **#include**d more than once when translating a source file (it is a requirement that implementations support this usage for standard headers). The following are some of the techniques implementations use to reduce the overhead of subsequent **#include**s.

- A common convention is to bracket the contents of a header, starting with the preprocessing token sequence `#ifndef __H_file_name__/#define __H_file_name__` and ending with `#endif`. The processing of subsequent **#include**s of the same header is then reduced to the minimal processing needed to skip to the matching **#endif**. Some implementations (e.g., `gcc`) go one step further and detect headers that contain such bracketing the first time they are processed, and completely skips opening and processing the header if it is subsequently encountered again in a **#include** directive.
- Support the preprocessing directive **#import**.[353] This directive is equivalent to the **#include** directive except that if the specified header has already been included it is not included again.

Coding Guidelines

Some coding guideline documents recommend that implementation supplied headers appear before developer written headers, in a source file. Such recommendations overlook the possibility that a developer written header might itself **#include** an implementation header.

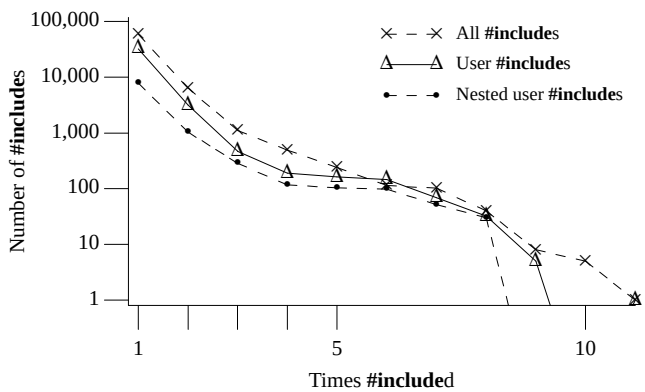


Figure 1896.1: Number of times the same header was **#include**d during the translation of a single translation unit. The crosses denote all headers (i.e., all systems headers are counted), triangles denote all headers delimited by quotes (i.e., likely to be user defined headers) and bullets denote all quote delimited headers **#include** nested at least three levels deep. Based on the translated form of this book's benchmark programs.

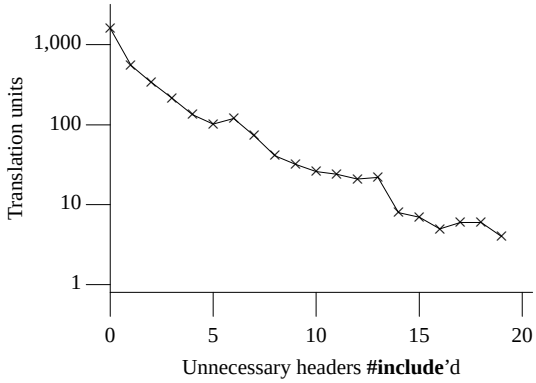


Figure 1896.2: Number of preprocessing translation units (excluding system headers) containing a given number of **#include**s whose contents are not referenced during translation (excludes the case where the same header is **#include**d more than once, see Figure 1896.1). Based on the translated form of this book's benchmark programs.

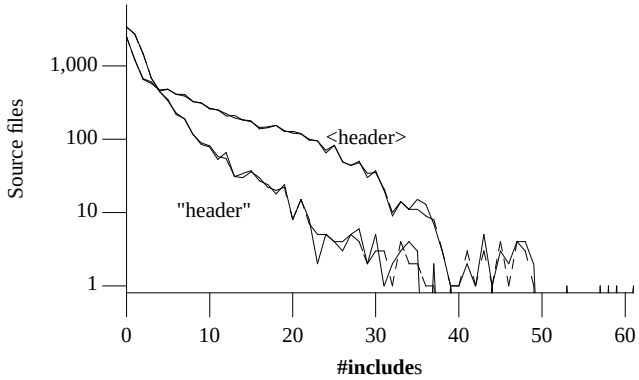


Figure 1896.3: Number of `.c` source files containing a given number of **#include** directives (dashed lines represent number of unique headers). Based on the visible form of the `.c` files.

Experience suggests that once a **#include** directive appears in a source file it is rarely removed (see Figure 1896.2) and that new **#include** directives are simply added after the last one. The issue of redundant code is discussed elsewhere.

redundant code 190

There does not appear to be a worthwhile benefit in ordering **#include** directives in any way (apart from any relative ordering dictated by dependencies between headers).

Table 1896.1: Occurrence of two forms of *header-name*s (as a percentage of all **#include** directives), the percentage of each kind that specifies a path to the header file, and number of absolute paths specified. Based on the visible form of the `.c` files.

Header Form	% Occurrence	% Uses Path	Number Absolute Paths
<h-char-sequence>	75.0	86.4	0
"q-char-sequence"	25.0	17.2	0

Semantics

#include h-char-sequence

A preprocessing directive of the form


```
# include <h-char-sequence> new-line
```

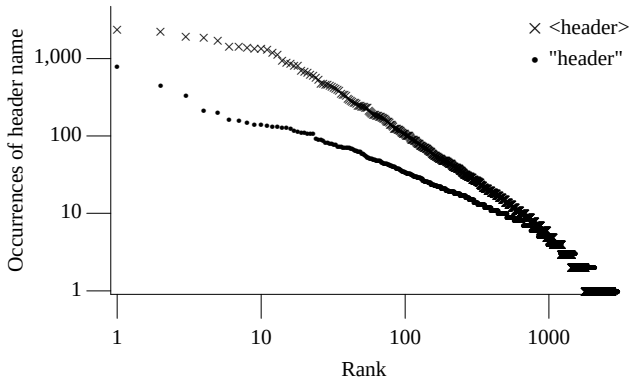


Figure 1896.4: *header-name* rank (based on character sequences appearing in **#include** directives) plotted against the number of occurrences of each character sequence. Also see Figure 792.26. Based on the visible form of the `.c` files.

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header.

Commentary

File systems invariably provide a unique method of identifying every file they contain (e.g., a *full path name*). The base document recognized the disadvantages of requiring that the full path name be specified in each **#include** directive and permitted a substring of it to be given. The implementation-defined places are usually additional character sequences (e.g., directory names) added to the *h-char-sequence* in an attempt to create a full path name that refers to an existing file.

918 header name syntax

Rationale

The file search rules used for the filename in the **#include** directive were left as implementation-defined. The Standard intends that the rules which are eventually provided by the implementor correspond as closely as possible to the original K&R rules. The primary reason that explicit rules were not included in the Standard is the infeasibility of describing a portable file system structure. It was considered unacceptable to include UNIX-like directory rules due to significant differences between this structure and other popular commercial file system structures.

Nested include files raise an issue of interpreting the file search rules. In UNIX C a **#include** directive found within an included file entails a search for the named file relative to the file system *directory* that holds the outer **#include**. Other implementations, including the earlier UNIX C described in K&R, always search relative to the same *current directory*. The C89 Committee decided in principle in favor of K&R approach, but was unable to provide explicit search rules as explained above.

Other Languages

Other languages (or an extension provided by their implementations) commonly use the double-quote delimited form.

Common Implementations

The character sequence between the < and > delimiters is invariably treated as the name of a file, possibly including a path. The ordering of the search sequence used for directives having the form *<h-char-sequence>* is often different from that used for the form *"q-char-sequence"*. For instance, in the *<h-char-sequence>* case the contents of `/usr/include` might be searched first, followed by the contents of the directory containing the `.c` file, while in *"q-char-sequence"* case the contents of the directory containing the `.c` file might be searched first, followed by other places.

1909 #include mapping to host file

The environment in which a translator executes may also affect the sequence of places that are searched. For instance, the affect of relative path names (e.g., `../proj/abc.h`) on the identity of the current directory.

gcc searches two directories, /usr/include and another directory that holds very machine specific files, such as stdarg.h (e.g., /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include on your authors computer). gcc supports the **#include_next** directive. This directive causes the search algorithm to skip some of the initial implementation-defined places that would normally be searched. The initial places that are skipped are those that were searched in locating the file containing the **#include_next** directive (including the place where the search succeeded).

Tzerpos and Holt[1383] describe a well-formedness theory of header inclusion that enables unnecessary **#include** directives to be deduced.

Coding Guidelines

The standard does not specify the order in which the implementation-defined places are searched. This is a potential coding guideline issue because it is possible that a *h-char-sequence* will match in more than one of the places (i.e., there is a file having the same name along several of the different possible search paths). The behavior is thus dependent (i.e., it is assumed that the contents of the different headers will be different) on the order in which the places are searched.

Experience suggests that the affect of a translator locating an **#include**d file different from the one expected to be located by the developer has one of two consequences— (1) when the contents of the file accessed is similar to the one intended (e.g., a different version of the intended file) the source file may be successfully translated, and (2) when the contents of the file accessed has no connection with the intended file the source is rarely successfully translated. The problem might therefore be considered to be one of version management, rather than the choice of characters used in a *h-char-sequence*. There are a number of reasons why a solution to this issue is to not use *h-char-sequence*s at all, including the following:

#include 1898 places to search for

- For the < > delimited form, implementations usually look in a predefined location first (as described in the Common implementation section above and in the following C sentence). Ensuring that the names chosen by developers for the headers they create are different from those of system headers is an almost impossible task. While it might be possible to enumerate the set of names of existing file names of system headers contained in commercially important environments, members are likely to be added to this set on a regular basis. Rather than trying to avoid using file names likely to match those of system headers, developers could ensure that places containing system headers are searched last.

- The < > delimited form is often considered to denote *externally* supplied headers (e.g., provided by the implementation or translator environment vendor). What constitutes a system supplied header is open to interpretation. One distinction that can be made between system and developer headers is that developers do not control of the contents of system headers. Consequently, it can be argued that their contents are not subject to coding guidelines. Headers whose contents have been written by developers are subject to coding guidelines. The convention generally adopted to indicate this status is to use the double-quote character delimit form of **#include**.

Rev 1897.1 Developer written headers in a **#include** directive shall not be delimited by the < and > characters.

Developers sometimes specify full path names in headers (see Table 1896.1). This is a configuration management issue and is not considered to be within the scope these coding guidelines.

Table 1897.1: Number of various kinds of identifiers declared in the headers contained in the `/usr/include` directory of some translation environments. Information was automatically extracted and represents an approximate lower bound. Versions of the translation environments from approximately the same year (mid 1990s) were used. The counts for ISO C assumes that the minimum set of required identifiers are declared and excludes the type generic macros.

Information	Linux 2.0	AIX on RS/6000	HP/UX 9	SunOS 4	Solaris 2	ISO C
Number of headers	2,006	1,514	1,264	987	1,495	24
macro definitions	10,252	18,637	13,314	11,987	10,903	446
identifiers with external linkage	1,672	1,542	1,935	616	1,281	487
identifiers with internal linkage	80	34	2012	0	5	0
tag declaration	716	1,088	899	1,208	945	3
typedef name declared	1,024	828	15	493	1,027	55

1898 How the places are specified or the header identified is implementation-defined.

#include places to search for

Commentary

The differences between the environments in which translation occurs has narrowed over the years. However, even although there may be much common practice, such are issues are considered to be outside the scope of the C Standard.

10 program transformation mechanism

Common Implementations

Implementations invariably search one or more predefined locations first (e.g., `/usr/include`), followed by a list of alternative places. A number of techniques are used to allow developers to specify a list of alternative places to be searched for files corresponding to the headers specified in a **`#include`** directive. For instance, the alternative places may be specified via a translator command line option (e.g., `-I`), in a translator configuration file (e.g., `gcc` version 2.91.66 hosted on RedHat Linux reads many default from the file `/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs`, although the path `/usr/include` is still hard coded in the translator sources), or an environment variable (e.g., several Microsoft windows based translators use `INCLUDE`).

The directory separator used in Unix and MS-DOS slants in different directions. Many implementations, in both environments, recognize both characters as directory delimiters. One consequence of this is that escape sequences are not recognized as such (something that is unlikely to be a problem in header names).

The RISCOS environment does not support filenames ending in `.h`. The implementation-defined behavior for this host is to look in a directory called `h`, for a file of the given name with the `.h` removed.

Coding Guidelines

The implementation-defined behavior associated with how the places are specified occurs outside of the source code and is the remit of configuration management guidelines. For this reason nothing further is said here.

1899 A preprocessing directive of the form

#include q-char-sequence

```
# include "q-char-sequence" new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters.

Commentary

The commonly accepted intent of this form of the **`#include`** directive is that it is used to reference source files created by developers (i.e., headers that are not provided as part of the implementation or host environment). The only syntactic difference between *`q-char-sequence`* and *`h-char-sequence`* is that neither sequence may contain their respective delimiters.

918 header name syntax

Most *`q-char-sequence`*s end with one of two character sequences (i.e., `.c` or `.h`). The character sequences before these suffixes is often called the *header name*.

Other Languages

The use of double-quote as the delimiter is the almost universal form used in other languages (although some use the ' character because that is what is used to delimit string literals).

Coding Guidelines

The term commonly used to refer to these source files is *header*. The context of the conversation often being used to distinguish any other intended usage. The intent is that the contents of these source files is controlled by developers and as such they are subject to coding guidelines.

The named source file is searched for in an implementation-defined manner. 1900

Commentary

While this "implementation-defined manner" might be the same as that for the < > delimited form. The intent is for it to be sufficiently different that developers do not need to be concerned about the name of a header created by them matching one provided as part of the implementation (and therefore potentially found by the translator when searching for a matching header). For instance, your author does not know the names of most of the 304 files (e.g., `compface.h`) contained in `/usr/include` on his software development computer. The discussion on the < > delimited form is applicable here.

#include 1897 h-char-sequence

Common Implementations

The search algorithm used invariably differs from that used for the < > delimited form (otherwise there would be little point in distinguishing the two cases). The search algorithm used by some implementations is to first look in the directory containing the source file currently being translated (which may itself have been included). If that search fails, and the current source file has itself been included, the directory containing the source file that **`#include`** it is then searched. This process continuing back through any nested **`#include`** directives. For instance, in:

file_1.c
```
#include "abc.h"
```

file_2.c
```
#include "/foo/file_1.c"
```

file_3.c
```
#include "/another/path/file_2.c"
```

(assuming the translation environment supports the path names used), translating the source file `file_3.c` causes `file_2.c` to be included, which in turn includes `file_3.c`. The source file `abc.h` will be searched for in the directories `/foo`, `/another/path` and then the directory containing `file_3.c`.

Some implementations use the double-quote delimited form within their system headers, to change the default first location that is searched. For instance, a third-party API may contain the header `abc.h`, which in turn needs to include `ayx.h`. Using the form `"ayx.h"` means that the implementation will search in the directory containing `abc.h` first, not `/usr/include`. This usage can help localize the files that belong to specific APIs. Other implementations use a search algorithm that starts with the directory containing the original source file being translated.

If the source file is not found after these places have been searched, some implementations then search other places specified via any translator options. Other implementations simply follow the behavior described by the following C sentence (which has the consequence of eventually checking these other places).

#include 1898 places to search for

If this search is not supported, or if the search fails, the directive is reprocessed as if it read 1901

`#` **`include`** <*h-char-sequence*> *new-line*

with the identical contained sequence (including > characters, if any) from the original directive.

Commentary

The previous search can fail in the sense that it does not find a matching source file.

Some existing code uses the double-quote delimited form of **#include** directive to include headers provided by the implementation (rather than the < > delimited form). This requirement ensures that such code continues to be conforming.

1902 144) As indicated by the syntax, a preprocessing token shall not follow a **#else** or **#endif** directive before the terminating new-line character.

footnote 144

Commentary

Saying in words what is specified in the syntax.

Common Implementations

Many early implementations (and some present days ones, for compatibility with existing source) treated any sequence of characters following one of these directives as a comment, e.g., `#endif X == 1`.

1903 However, comments may appear anywhere in a source file, including within a preprocessing directive.

Commentary

A comment is replaced by a single space character prior to preprocessing.

126 comment replaced by space
1858 preprocessing directive ended by

1904 A preprocessing directive of the form

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted.

Commentary

This form permits the < > or double-quote delimited forms to be generated via macro expansion. However, it is rarely used (11 instances in over 60,000 **#include** directives in the visible source of the `.c` files). Whether this is because developers are unaware of its existence, or because it has little utility is not known.

1914 #include example 2

1905 The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.)

#include macros expanded

Commentary

To be exact, the preprocessing tokens after **include** in the directive up to the first new-line character are processed just as in normal text.

1906 (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.)

Commentary

This C sentence provides explicitly clarification that macro replacement occurs in this case (the same clarification is also given elsewhere).

1991 #line macros expanded

1907 The directive resulting after all replacements shall match one of the two previous forms.[145)]

Commentary

It is not a violation of syntax if the directive does not match one of the two previous forms, because the syntax of this form has been matched. It is a violation of semantics and therefore the behavior is undefined.

1908 The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

Commentary

This implementation-defined behavior may take a number of forms, including:

1958 operator
1963 if result not valid

- The ## operator can be used to glue preprocessing tokens together. However, the behavior is undefined if the resulting character sequence is not a valid preprocessing token. For instance, the five preprocessing tokens {{} {string} {.} {h} {}} cannot be glued together to form a valid preprocessing token without going through intermediate stages whose behavior is undefined.

1950 operator

- Creating a preprocessing token, via macro expansion, having the double-quote delimited form (i.e., a string preprocessing token) need not depend on any implementation-defined behavior. The stringize operator can be used to create a string preprocessing token.

- Other implementation-defined behaviors might include the handling of space characters. For instance, in the following:

```
1  #define bra <
2  #define ket >
3  #include bra stdio.h ket
```

  does the implementation strip off the space character at the ends of the delimited character sequence?

Coding Guidelines

Given the rarity of use of this form of **#include** no guideline recommendations are given here.

Example

```
1  #define mk_sys_hdr(name) < ## name ## >
2
3  #if BUG_FIX
4  #define VERSION 2a /* works because pp-numbers include alphabetics */
5  #else
6  #define VERSION 2
7  #endif
8
9  #define add_quotes(a) # a
10 #define mk_str(str, ver) add_quotes(str ## ver)
11
12 #include mk_str(Version, VERSION)
```

#include mapping to host file

The implementation shall provide unique mappings for sequences consisting of one or more ~~letters or digits (as defined in 5.2.1)~~ nondigits or digits (6.4.2.1) followed by a period (.) and a single ~~letter~~ nondigit. 1909

Commentary

This C sentence and the following ones in this C paragraph are a specification of the minimum set of requirements that an implementation must meet. For sequences outside of this set the implementation mapping may be non-unique (like, for instance, the Microsoft Windows technique of mapping files ending in .html to .htm). The handling of character sequences that resemble UCNs may also differ, e.g., "\ubada\file.txt" (Ubada is a city in Tanzania and BADA is the Hangul symbol 붚 in ISO 10646). The standard does not permit any number of period characters because many operating systems do not permit them (at least one, RISCOS, does not permit any).

The wording was changed by the response to DR #302 to extend the specification to be more consistent with C++.

C++

The implementation provides unique mappings for sequences consisting of one or more nondigits (2.10) followed by a period (.) and a single nondigit. 16.2p5

Other Languages

Other languages either specified to operate within the same operating systems and file systems limitations as C and as such have to deal with the same issues, or require an integrated development environment to be created before they can be used.

Common Implementations

Implementations invariably pass the sequence of characters that appear between the delimiters (when searching other places a directory path may be added) as an argument in a call to `fopen` or equivalent system function. The called library function will eventually call some host operating system function that interfaces to the host file system. The C translator's behavior is thus controlled by the characteristics of the host file system and how it maps character sequences to file names. The handling of the period character varies between file systems, known behaviors include:

- Unix based file systems permit more than one period in a file name.
- MS-DOS based file systems only permit a single period in a file name.
- RISCOS, an operating system for the Acorn ARM processor does not support filenames that contain a period. For this host file names, that contained a period, specified in a **`#include`** directive were mapped using a directory structure. All file names ending in the characters `.h` were searched for in a directory called `h`.

Coding Guidelines

Because an implementation is not required to provide a unique mapping for all sequences it is possible that an unintended header or source file will be accessed, or the translator will fail to identify a known header or source file. The possible consequences of an unintended access are discussed elsewhere, while failure to identify known header or source file will cause a diagnostic to be issued. The cost/benefit issues associated with using character sequences having a unique mapping in the different environments that the source may be translated in is outside the scope of these coding guidelines.

1897 #include h-char-sequence
1896 source file inclusion

1910 The first character shall ~~be a letter~~ not be a digit.

Commentary

This requirement only applies to the first character of the sequence that implementations are required to provide a unique mapping for.

The wording was changed by the response to DR #302.

C90

The requirement that the first character not be a digit is new in C99. Given that it is more restrictive than that required for existing C90 implementations (and thus existing code) it is unlikely that existing code will be affected by this requirement.

C++

This requirement is new in C99 and is not specified in the C++ Standard (the argument given in the C90 subsection (above) also applies to C++).

Common Implementations

Most implementations support a first character that is not a letter.

header name significant characters

The implementation may ignore the distinctions of alphabetical case and restrict the mapping to eight significant characters before the period. 1911

Commentary

These permissions reflect known characteristics of file systems in which translators are executed.

C90

The limit specified by the C90 Standard was six significant characters. However, implementations invariably used the number of significant characters available in the host file system (i.e., they do not artificially limit the number of significant characters). It is unlikely that a header of source file will fail to be identified because of a difference in what used to be a non-significant character.

C++

The C++ Standard does not give implementations any permissions to restrict the number of significant characters before the period (16.1p5). However, the limits of the file system used during translation are likely to be the same for both C and C++ implementations and consequently no difference is listed here.

Common Implementations

All file systems place some limits on the number of characters in a source file name— for instance:

- Most versions of the Microsoft DOS environment ignore the distinction of alphabetic case and restrict the mapping to eight significant characters before any period (and a maximum of three after it).
- POSIX requires that at least 14 characters be significant in a file name (it also requires implementations to support at least 255 characters in a pathname). Many Linux file systems support up to 255 characters in a filename and 4095 characters in a pathname.

Coding Guidelines

The potential problems associated with limits on sequences characters that are likely to be treated as unique is a configuration management issue that is outside the scope of these coding guidelines.

A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1). 1912

Commentary

Thus **#include** directives can be nested within source files whose contents have themselves been **#include**d. This issue is discussed elsewhere. While this permission only applies to source files, an implementation using some form of precompiled headers (which are not source files within the standard's definition of the term) that did not support this functionality would not be popular with developers.

limit 295
#include nesting
header 121
precompiled
source files 108

EXAMPLE 1 The most common uses of **#include** preprocessing directives are as in the following: 1913

```
#include <stdio.h>
#include "myprog.h"
```

Other Languages

Some languages only have a single form of **#include** directive for all headers.

#include example 2

EXAMPLE 2 This illustrates macro-replaced **#include** directives: 1914

```
#if VERSION == 1
        #define INCFILE  "vers1.h"
#elif VERSION == 2
        #define INCFILE  "vers2.h"   // and so on
```

```
#else
        #define INCFILE  "versN.h"
#endif
#include INCFILE
```

Commentary

This example does not illustrate any benefit compared to that obtained from placing separate **#include** directives in each arm of the conditional inclusion directive.

1915 **Forward references:** macro replacement (6.10.3).

1916 145) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2);

footnote 145

Commentary

String concatenation occurs in translation phase 6 and so it is not possible to join together two existing strings to form another string within a **#include** directive.

135 translation phase 6

1917 thus, an expansion that results in two string literals is an invalid directive.

Commentary

It is an invalid directive in that it violates a semantic requirement and thus the behavior is undefined. It is not a syntax violation.

6.10.3 Macro replacement

Constraints

macro replacement

1918 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

replacement list identical if

Commentary

This is actually a definition in a Constraints clause (it is used by two constraints in this C subsection).

The check against same spelling only needs to take into account the significant characters of an identifier. Considering all white-space separations to be identical removes the need for developers to be concerned about use of different source layout (e.g., indentation) and method of spacing (e.g., space character vs. horizontal tab).

282 internal identifier significant characters

Rationale

The specification of macro definition and replacement in the Standard was based on these principles:

- Interfere with existing code as little as possible.
- Keep the preprocessing model simple and uniform.
- Allow macros to be used wherever functions can be.
- Define macro expansion such that it produces the same token sequence whether the macro calls appear in open text, in macro arguments, or in macro definitions.

Preprocessing is specified in such a way that it can be implemented either as a separate text-to-text prepass or as a token-oriented portion of the compiler itself. Thus, the preprocessing grammar is specified in terms of tokens.

object-like macro redefinition

An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. 1919

Commentary

There was an existing body of code, containing redefinitions of the same macro, when the C Standard was first written. The C committee did not want to specify that existing code containing such usage was non-conforming, but they did consider the case where the bodies of any subsequent definitions differed to be an erroneous usage.

EXAMPLE 1983
macro redefinition

C90

The wording in the C90 Standard was modified by the response to DR #089.

Common Implementations

#define/#undef stack

Some translators permit multiple definitions of a macro, independently of the contents of the contents of the bodies. The behavior is for a new definition to cause the previous body to be pushed, in a stack-like fashion. Any subsequent **#undef** of the macro name popping this stacked definition and to make it the current one.

Coding Guidelines

C permits more than one definition of the same macro name, with the same body, and more than one external definition of the same object, with the same type and the coding guideline issues are the same for both (in both cases translators are not always required to issue a diagnostic if the definitions are considered to be different).

linkage 420
identifier 422.1
declared in one file

In both cases a technique for avoiding duplicate definitions, during translation but not in the visible source, is to bracket definitions with `#ifndef MACRO_NAME/#endif` (in the case of the file scope object a macro name needs to be created and associated with its declaration). Using this technique has the disadvantage that it prevents the translator checking that any subsequent redeclarations of an identifier are the same (unless the bracketing occurs around the only textual declaration that occurs in any source file used to build a program).

function-like macro redefinition

Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical. 1920

Commentary

object-like 1919
macro redefinition

The issues are the same as for object-like macros, with the addition of checks on the parameters. Requiring that the parameters be spelled the same, rather than, for instance, that they have an identical effect, simplifies the similarity checking of two macro bodies. For instance, in:

```
1  #define FM(foo) ((foo) + x)
2  #define FM(bar) ((bar) + x)
```

a translator is not required to deduce that the two definitions of `FM` are structurally identical.

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro. 1921

Commentary

extended 216
character set

In the following (assuming `$` is a member of the extended character set and permitted in an identifier preprocessing token):

```
1  #define A$ x
```

an object-like macro with the name `A$` and the body `x` is defined, not macro with the name `A` and the body `$ x`.

There is no requirement that there be white-space following the **)** in a function-like macro definition.

C90

The response to DR #027 added the following requirements to the C90 Standard.

DR #027

Correction

Add to subclause 6.8, page 86 (Constraints):

*In the definition of an object-like macro, if the first character of a replacement list is not a character required by subclause 5.2.1, then there shall be white-space separation between the identifier and the replacement list.**

*[Footnote *: This allows an implementation to choose to interpret the directive:*

```
#define THIS$AND$THAT(a, b) ((a) + (b))
```

as defining a function-like macro THISANDTHAT, rather than an object-like macro THIS. Whichever choice it makes, it must also issue a diagnostic.]

However, the complex interaction between this specification and UCNs was debated during the C9X review process and it was decided to simplify the requirements to the current C99 form.

```
#define TEN.1 /* Define the macro TEN to have the body .1 in C90. */
              /* A constraint violation in C99. */
```

C++

The C++ Standard specifies the same behavior as the C90 Standard.

Common Implementations

HP–was DEC– treats $ as part of the spelling of the macro name.

1922 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition.

Commentary

This requirement is the macro invocation equivalent of the one for function calls.

998 function call arguments agree with parameters

C90

If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined.

The behavior of the following was discussed in DR #003q3, DR #153, and raised against C99 in DR #259 (no committee response was felt necessary).

```
#define foo() A
#define bar(B) B

foo() // no arguments
bar() // one empty argument?
```

What was undefined behavior in C90 (an empty argument) is now explicitly supported in C99. The two most likely C90 translator undefined behaviors are either to support them (existing source developed using such a translator will may contain empty arguments in a macro invocation), or to issue a diagnostic (existing source developed using such a translator will not contain any empty arguments in a macro invocation).

C++

The C++ Standard contains the same wording as the C90 Standard.

C++ translators are not required to correctly process source containing macro invocations having any empty arguments.

Common Implementations

Some C90 implementations (e.g., `gcc`) treated empty arguments as an argument containing no preprocessing tokens, while others (e.g., Microsoft C) treated an empty argument as being a missing argument (i.e., a constraint violation).

... arguments macro

Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the `...`). 1923

Commentary

Rationale

There must be at least one argument to match the ellipsis. This requirement avoids the problems that occur when the trailing arguments are included in a list of arguments to another macro or function. For example, if `dprintf` had been defined as

```
#define dprintf(format,...) \
        dfprintf(stderr, format, __VA_ARGS__)
```

and it were allowed for there to be only one argument, then there would be a trailing comma in the expanded form. While some implementations have used various notations or conventions to work around this problem, the Committee felt it better to avoid the problem altogether.

C90

Support for the form `...` is new in C99.

C++

Support for the form `...` is new in C99 and is not specified in the C++ Standard.

Common Implementations

`gcc` allowed zero arguments to match a macro parameter defined using the `...` form.

Coding Guidelines

While some developers may be confused because the requirements on the number of arguments are different from functions defined using the ellipsis notation, passing too few arguments is a constraint violation (i.e., translators are required to issue a diagnostic that a developer then needs to correct).

---

macro invocation) terminates it

There shall exist a `)` preprocessing token that terminates the invocation. 1924

Commentary

While this requirement is specified in the syntax, it is interpreted as requiring the `)` preprocessing token to occur before any macro replacement of the identifiers following the matching `(` preprocessing token. For instance, in:

```
1  #define R_PAREN )
2
3  #define FUNC(a) a
4
5  static int glob = (1 + FUNC(1 R_PAREN );
```

the invocation is terminated by the `)` preprocessing token that occurs immediately before `;`, not the expanded form of `R_PAREN`.

1925 The identifier **__VA_ARGS__** shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the ~~arguments~~parameters.

Commentary

This requirement simplifies a translators processing of occurrences of the identifier **__VA_ARGS__**.

This typographical correction was made by the response to DR #234.

C90

Support for **__VA_ARGS__** is new in C99.

Source code declaring an identifier with the spelling __VA_ARGS__ will cause a C99 translator to issue a diagnostic (the behavior was undefined in C90).

C++

Support for **__VA_ARGS__** is new in C99 and is not specified in the C++ Standard.

Common Implementations

gcc required developers to give a name to the parameter that accepted a variable number of arguments. This parameter name appeared in the replacement list wherever the variable number of arguments were to be substituted.

Example

```
1  /*
2   * The following are constraint violations.
3   */
4  #define __VA_ARGS__
5  #define jparks __VA_ARGS__
6  #define jparks(__VA_ARGS__)
7  #define jparks(__VA_ARGS__, ...) __VA_ARGS__
8
9  #define jparks(x) x
10       jparks(__VA_ARGS__)
11
12 #define jparks(x, ...) x
13       jparks(__VA_ARGS__,1)
14 /*
15  * The following break the spirit, if not the wording
16  * of this constraint.
17  */
18 #define jparks(x, y) x##y
19       jparks(__VA, _ARGS__)
20
21 #define jparks(x, y, ...) x##y
22       jparks(__VA, _ARGS__, 1)
```

1926 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

macro parameter unique in scope

Commentary

This constraint is the macro equivalent of the one given for objects with no linkage. Its scope is the list of parameters in the macro definition and the body of that definition. This scope ends at the new-line that terminates the directive. Macro parameters are also discussed elsewhere.

1350 declaration only one if no linkage
1934 macro parameter scope extends
396 identifier macro parameter

Semantics

1927 The identifier immediately following the **define** is called the *macro name*.

macro name identifier

Commentary

This defines the term *macro name*. This term is generically used in software engineering to refer to this kind of entity.

---

macro one name space

There is one name space for macro names. 1928

Commentary

Object-like and function-like macro names exist in the same name space. However, an identifier defined as a function-like macro is only treated as such when its name is followed by an opening parenthesis. Name spaces are also discussed elsewhere.

function-like macro followed by (1935
name space 438

---

white-space before/after replacement list

Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro. 1929

Commentary

Specifying that such white-space should be considered to part of the replacement list has potential maintenance and comprehension costs (it restricts how the start of the replacement list may be indented and white-space following the replacement list is not immediately visible to readers) for no obvious benefit.

Example

In the following the string literal "_ _" is assigned to p.

```
#define str_ize(a)  #a
#define M  _ _

char *p = str_ize(M);
```

If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement. 1930

Commentary

tokens in directive not expanded unless 1867

This is a special case of a more general specification given elsewhere.

Common Implementations

Some preprocessors used to perform this kind of replacement (some past entries in the Obfuscated C contest[43] relied on such translator behavior).

Example

In the following, even although the identifier `define` is defined as a macro, the line starting **`#define`** still processed as a macro definition directive, and not as a **`#undef`** directive.

```
#define define undef

#define X Y
```

---

macro object-like

A preprocessing directive of the form 1931

`#` **`define`** *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[146)] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

Commentary

This defines the term *object-like* macro. This term is not commonly used by developers, who tend to use the generic term *macro* for all macro definitions and when a distinction needs to be made use the term *function macro* (rather than the technically correct term *function-like macro*) to refer to the case of a macro defined to have parameters. A macro's replacement list is commonly known as a *macro body*, or simply its *body*.

The preprocessing tokens in a `text-line` are unconditional scanned for instances of macro names to expand, as are preprocessing tokens in some preprocessing directives.

1854 preprocessor directives syntax

The standard lists a few restrictions on identifiers that can be defined as macro names. The issue of implementation limits on the number of macros that may be defined in one preprocessing translation unit is discussed elsewhere.

2026 predefined macros not #defined
287 limit macro definitions

Other Languages

Some languages use **`def`** rather than **`define`**.

Common Implementations

Some preprocessors have a maximum limit on the number of characters that can occur in a replacement list (e.g., an early version of Microsoft C[932] had a 512 byte limit; a limit of 4096 is still seen in some preprocessors).

Implementations invariably provide a mechanism that is external to the source code for defining macros, e.g., the `-D` command line option.

Coding Guidelines

Macros can be defined to serve a variety of purposes (see Table 1931.1 for measurements of actual usage) including:

- Giving a symbolic name to a constant or expression. The issue of symbolic names is discussed elsewhere, as are the advantages of using enumeration types for related identifiers.
- Representing an expression without the overhead of a function call. Having made the decision to represent an expression with a symbolic name the decision on whether to use a function call or macro then needs to be made, the human decision making factors involved are discussed elsewhere.
- Parameterized code duplication. This kind of usage occurs because a function definition does not provide the necessary flexibility (for instance, the parameterization may involve constructs other than expressions).
- Parameterizing the definition of a type. This issue is discussed in more detail under typedef names.
- Controlling conditional inclusion. In this case their status as a macro definition is used as a flag.

822 symbolic name
517 enumeration set of named constants
0 agenda effects decision making
1629 typedef name syntax
476 boolean role

Some coding guideline documents recommend against what are sometimes known as *syntax changing* macro names. This terminology comes from the fact that uses of such macro names change the syntax, at least visually, of C. For instance, a developer familiar with Pascal might define the macro names `begin` and end to represent the C punctuators **`{`** and **`}`** respectively (this existing usage was one reason these macro names were not used as alternative spellings, in **`<iso646.h>`**, for these punctuators; it could have rendered existing conforming code nonconforming), or a developer wanting to modify existing code to use greater floating-point precision might define the macro name `float` to be **`double`**.

The growth in the usage languages with C-like syntax over the last 10 years means that these days it is rare for developers to attempt to change the visual appearance of C source to be closer to a language they are more familiar with. While a macro name that maps to a C token may be surprising to readers of the source, it is unlikely to conflict with their existing C knowledge, and therefore might be considered at worse a minor inconvenience (i.e., cost).

Defining a macro whose name is the same as a keyword means that the behavior of translated source can be very different from that expected from its visual appearance (such usage also results in undefined behavior

if the definition occurs prior to the inclusion of any library header). The presence of such a definition requires that readers substitute their existing, default response, knowledge of behavior for a new behavior (assuming that they had noticed the definition of the macro). Experience suggests that the short-term benefit of defining and using such macro names is less than the longer term (which may be only a few days) costs associated with comprehension and miscomprehension of the affected source.

Cg 1931.1 A source file shall not define a macro name to have the spelling of a keyword.

Replacement lists may look innocuous enough when viewed in isolation. However, in the context in which they occur the expanded form may interact in unexpected ways with adjacent tokens. For instance, looking at the components of the following source in isolation:

```
1  #define SUM  a + b
2
3  extern int glob;
4
5  void f(void)
6  {
7  int loc = glob * SUM;
8  }
```

the appearance of the replacement list of `SUM` suggests that `a` will be added to `b` and looking at the use of `SUM` in the initialization of `loc` suggests that it will be multiplied by the value of `glob`. However, the token sequence after macro replacement is `glob*a+b`, which has a very different interpretation.

The visual appearance of a replacement list containing statements can also be misleading. For instance, in:

```
1  #define INIT c=0; d=0;
2
3  extern int glob;
4
5  void f(void)
6  {
7  if (glob == 0)
8     INIT;
9  }
```

the assignment to `d` is not dependent on the value of `glob`. Which is counter to what the visual appearance of the source suggests.

A general solution to both of these problems is to bracket the replacement list, ensuring that the visually expected behavior is the same as the behavior that occurs after macro replacement.

Cg 1931.2 A replacement list having the form of an expression containing one or more binary operators shall be bracketed with parentheses, unless the binary operators are only those included in the production of a *postfix-expr*.

Cg 1931.3 A replacement list consisting of more than one statement shall be completely enclosed in a pair of braces (which make take the form of a **do** statement).

The visual appearance of declarations can also be deceptive when macro replacements are involved. For instance, in:

```
#define INFO_PTR int *

INFO_PTR glob_1,
         glob_2;
```

`glob_1` is declared to have a pointer type, while `glob_2` is declared to have an integer type.

The bracketing technique cannot be used with a replacement list that represents a type (it would violate C syntax). However, using a typedef name is not a general solution, it is possible to use macro names in situations where a typedef name cannot be used. For instance, in:

```
#define X_TYPE int

unsigned X_TYPE glob;
```

it is possible to modify the type denoted by `X_TYPE` because the macro expanded form represents a valid integer type when preceded by **`unsigned`**. However, the type denoted by a typedef name cannot be so modified.

1382 type specifiers possible sets of

Cg 1931.4

A replacement list shall not consist of a sequence of preprocessing tokens that has, after expansion, the syntax of a pointer type.

The replacement list of a macro definition has to appear on a single logical source line. Experience suggests that constructs that appear on separate lines in other contexts often appear on the same line within a replacement list. The developer cost (typing the characters) of using splicing, to give the replacement list a visible form that closely resembles that seen when it appears in other contexts is small. The benefit for subsequent readers is the ability to use the same strategies to read source constructs as they use in other contexts.

118 logical source line

118 physical source line

There are a number of ways in which token sequences appearing in various contexts might visually resemble each other. For instance, in the following definitions both `ZERO_ARRAY_1` and `ZERO_ARRAY_2` visually associate preprocessing tokens in the macro body, while in `ZERO_ARRAY_3` preprocessing tokens in the macro body visual interacts with the preprocessing tokens in the preprocessing directive.

```
#define ZERO_ARRAY_1(a, n) for (int i = 0; i < (n); i++) \
                               (a)[i]=0;
#define ZERO_ARRAY_2(a, n)       \
   for (int i = 0; i < (n); i++) \
       (a)[i]=0;
#define ZERO_ARRAY_3(a, n) for (int i = 0; i < (n); i++) \
   (a)[i]=0;
```

The following guideline recommendation leaves the decision on what constitutes *the same visual layout* to developers.

Rev 1931.5

Token sequences shall have the same visual layout in the replacement list of a macro definition as they do in other contexts.

Usage

Usage information on the number of macro names defined in source files is given elsewhere.

287 limit macro definitions

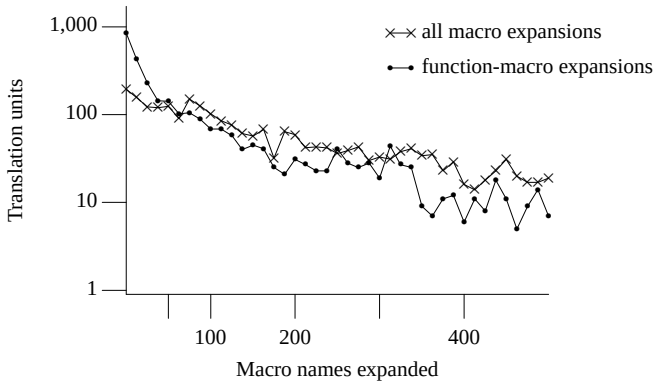


Figure 1931.1: Number of translation units containing a given number of macro names which were macro expanded, excluding expansions that occurred while processing the contents of system headers. Based on the translated form of this book's benchmark programs.

Table 1931.1: Detailed breakdown of the kinds of replacement lists occurring in macro definitions. Adapted from Ernst, Badros, and Notkin.[396]

Replacement List	%	Example
constant	42	`#define ARG_MAX 1000`
expression	33	`#define SHFT_UP(x) ((x) << 8)`
empty	6.9	`#define DUMMY`
unknown identifier	6.9	`#define INTERN_BUF buffer`
statement	5.1	`#define TERMINATE goto func_end`
type	2.1	`#define NODE_PTR void *`
other	1.9	`#define OPTION -X=23`
symbol	1.4	`#define ALLOC_STORAGE malloc`
syntactic	0.5	`#define begin {`

Table 1931.2: Common macro definitions listed with an abstracted form of their replacement list (as a percentage of all macro definitions). Note that *function-call* may also be a macro invocation. Based on the visible form of the `.c` and `.h` files.

Kind of Macro Defined and Abstract Form of its Replacement List	%
object-like macro *integer-constant*	50.7
object-like macro identifier	5.9
object-like macro expression	5.8
function-like macro function-call	4.7
object-like macro function-call	3.7
object-like macro *string-literal*	3.4
function-like macro expression	3.4
object-like macro	3.4
object-like macro *constant-expression*	2.0
function-like macro	1.7
others	15.4

The replacement list is then rescanned for more macro names as specified below. 1932

Commentary

This sentence was added by the response to DR #306 and removes the possibility of a reader interpreting the rescanning clause as only applying to function-like macros.

rescanning 1968
macro 1933
function-like

macro
function-like

A preprocessing directive of the form 1933

```
# define identifier lparen identifier-list_opt ) replacement-list new-line

# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a *function-like macro* with ~~arguments,~~ parameters, whose use is similar syntactically to a function call.

Commentary

This defines the term *function-like* macro. This term is commonly used by developers. Function-like macro definitions do not contain any type information. Replacement is based solely on matching preprocessing token spellings.

Limits on the number of parameters a function-like macro definition may contain are discussed elsewhere. [290] limit macro parameters

The wording was changed by the response to DR #307.

C90

Support for the `...` notation in function-like macro definitions is new in C99.

C++

Support for the `...` notation in function-like macro definitions is new in C99 and is not specified in the C++ Standard.

Common Implementations

`gcc` supported a form of `...` notation in its implementation of the C90 Standard.

Coding Guidelines

The guideline recommendations given in the C sentence for describing object-like macros are written in a general form, i.e., they also apply to function-like macro definitions. [1931] macro object-like

The visual appearance of a function-like macro's replacement list can be misleading in suggesting that an operation is performed on a parameter. For instance, in:

```
1  #define FUNC(x)  x * glob_1
2
3  extern int glob_1,
4             glob_2;
5
6  void f(void)
7  {
8  int loc = FUNC(2 + glob_2);
9  }
```

a reader looking at the replacement list for `FUNC` might believe that the macro expanded argument will be multiplied by `glob_1`. In fact the expanded token sequence has very different behavior, i.e., `2+glob_2*glob_1`.

A general solution is to use parenthesis to ensure that the actual behavior matches that expected from the visual appearance of the source.

Cg 1933.1

Any parameter of a function-like macro appearing as an operand in an expression shall be parenthesized, unless it is the operand of the # or ## operators.

macro parameter scope extends

The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. 1934

Commentary

The visibility of the parameters also extends over the entire replacement list and is not affected by any identifiers declared within the replacement list (they are simply treated as a sequence of preprocessing tokens until later phases of translation). Scope is extensively discussed elsewhere.

scope 390 of identifiers

Usage

Usage information on the number of parameters in function-like macro definitions is given elsewhere.

limit 290 macro parameters

function-like macro followed by (

Each subsequent instance of the function-like macro name followed by a **(** as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). 1935

Commentary

No formal syntax is specified for the sequence of preprocessing tokens that form an invocation of a function-like macro. However, in some contexts the sequence of preprocessing tokens in an invocation of a function-like macro may result in undefined behavior (e.g., preprocessing tokens having the form of a preprocessing), or the context in which the invocation occurs may have its own syntax (e.g., preprocessing directives are terminated by a new-line).

argument 1940 resemble preprocessing directive

preprocessing directive 1858 ended by

It is possible to suppress the expansion of a *function-like* macro by ensuring that it is not followed by a **(** preprocessing token (e.g., by enclosing the macro name in parenthesis):

```
#define FUNC(a) (a+1)

void f(void)
{
(FUNC)(3);     /* Final token sequence is (FUNC)(3) */
}
```

even if a **(** preprocessing token occurs in the sequence of preprocessing tokens at some point (DR #017q21):

```
#define L_PAREN (
#define F_M(a) a

char *p = F_M L_PAREN "abc" ); /* Expands to F_M("abc") not "abc" */
```

Coding Guidelines

Some implementations provide both function and macro definitions of some library functions. Developers wanting to ensure that the function's definitions are invoked parenthesize the name of the function to prevent it being treated as a function-like macro. An occurrence of an identifier currently defined as a function-like macro and not followed by a **(** preprocessing token could be a fault (in which case a translator diagnostic is likely to be generated because of a reference to an undeclared identifier), or the same identifier is used to define different entities (this issue is discussed elsewhere).

identifier 792 syntax

The replaced sequence of preprocessing tokens is terminated by the matching **)** preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. 1936

Commentary

The syntax of function-like macros does not specify which right parentheses terminates an argument list. Hence the need for this wording. Skipping intervening matched pairs of left and right parentheses preprocessing tokens allows arbitrary expressions, which may containing parentheses, to be passed as arguments. Any preprocessing tokens between the matched parentheses are treated as belonging to the argument and not part of the syntax of the macro invocation. For instance:

```
#define EXPR(x, y) x * y

int glob = EXPR( (a - b), (c == d) );
```

The) preprocessing token is searched for in the source file without performing macro expansion (DR #017q21):

```
#define i(x) 3
#define a i(yz
#define b )

a b ) ;
/*
 * Expands via the following stages:

 i(yz b )  ) delimits the argument list before b is expanded
 i(yz ) )  the argument to i is the two preprocessing tokens yz )
 3
 */
```

1937 Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

Commentary

This specification only applies outside of preprocessing directives. The intent is for function-like macro invocations to have the same flexibility in their visual layout as function calls.

1858 preprocessing directive ended by

Coding Guidelines

The issues associated with laying out source code are driven by its visible form, not its expanded form.

770 words white space between
940 expression visual layout
1348 declaration visual layout
1707 statement visual layout
1810 translation unit syntax

1938 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro.

Commentary

This introduces the common usage term *arguments* to refer to these preprocessing token sequences. Limits on the number of arguments that may appear in an invocation of function-like macro are discussed elsewhere.

macro arguments
291 limit arguments in macro invocation

1939 The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments.

macro arguments separated by comma

Commentary

The syntax of function-like macros does not specify which commas delimit the arguments. Hence the need for this wording. Skipping commas occurring between matched parentheses preprocessing tokens allows function call, which may themselves contain arguments, to be passed as arguments.

Coding Guidelines

An argument whose evaluation causes a side effect can sometimes result in program behavior that is surprising to developers (because they failed to take account of the argument being evaluated more than once). For instance, in the following fragment:

```
#define M(a, b)  ((a)*(b) + (a))

int glob = M(i++, j);
```

the object `i` is incremented twice. There are a number of possible guideline recommendations that prevent these surprises occurring, these include recommending that:

- The evaluation of arguments to macros not have side effects. Such a recommendation would require that developers be aware of whether an identifier followed by a left parentheses results in a macro replacement or a function call. At some future time a function call may be replaced by a macro invocation, which could then require that existing code be changed to ensure that arguments did not cause side effects (this goes against the aim that adherence to guideline recommendations not require an amount of effort that is out of proportion to the changes made to existing source). Side effect related issues in other language constructs are discusses elsewhere.

guideline recommendation 0
adherence has a reasonable cost
controlling expression 1740
if statement

- The expansion of a macro not result in a sequence of tokens that evaluate any of its arguments more than once. Syntactically it is possible to create a replacement list that follows this recommendation. However, semantically temporary variables of the correct type need to be visible and invoking the same macro twice in the same full expression is likely to result in undefined behavior.

```
1  #define M(a, b)  (t1=(a), t2=(b), (t1)*(t2) + (t1))
2
3  int glob = M(i++, j) + M(k, l);
```

  When using `gcc` this problem can be solved by making use of two extensions, (1) the **`typeof`** operator and (2) using the **`({ })`** punctuators to create an expression from a sequence of declarations and statements. For instance:

```
1  #define min(X, Y)               \
2          ({                      \
3          typeof (X) __x = (X),   \
4                     __y = (Y);   \
5          (__x < __y) ? __x : __y; \
6          })
```

The costs associated with both of these possible recommendations would be incurred for all function-like macros, while a benefit would only be obtained for a few uses. It would seem that neither of them has sufficient cost/benefit to make a guideline recommendation worthwhile.

Both of the previous possible recommendations treated the macro definition and its invocation in isolation. Recommending that an argument causing a side effect not be passed to a macro whose corresponding parameter is evaluated more than once is equivalent to one recommending that programs not contain faults, an issue that is discussed elsewhere.

guidelines 0
not faults

Example

```
1  #define M(x, y) /* ... */
2
3  M( f(1, 2), g((x=y++, y)))
4  M( {a=1 ; b=2;} ) /* A semicolon is not a comma         */
5  M( <, [ )         /* Passes the arguments < and [       */
6  M( (,), (...) )   /* Passes the arguments (,) and (...) */
7
8  #define START_END(start, end) start c=3; end
9
10 START_END( {a=1 , b=2;} ) /* braces are not parentheses */
11
12 /*
13  * To pass a comma token as an argument it is
14  * necessary to write:
15  */
16
17 #define COMMA ,
18
19 M(a COMMA b, (a, b))
```

1940 If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives[147], the behavior is undefined.

argument resemble preprocessing directive

Commentary

In the following code fragment:

```
#define FUNC_MACRO(x) /* ... */

FUNC_MACRO(
#define M 44

#if VAL == 1
          3
#else
          4
#endif
          )
```

possible behaviors include:

- Treating the sequence of preprocessing tokens between the matches parentheses treated as its argument.
- Treating those sequences of preprocessing tokens that have the form of a preprocessing directive as such a directive, i.e., defining `M` an object-like macro and passing either `3` or `4` as the argument to `FUNC_MACRO`.
- Issuing a diagnostic and failing to translate the source file.

Preprocessing directives can occur within the list of arguments in automatically generated, or processed, code. For instance, an expression original written by a developer may be expanded or split over several lines (source is often emailed and some email programs have a lower limit on the number of characters on a line than the C Standard).

Suppose line number 123 of a source file contained

```
dgay(some_very_long_and_complicated_expression_that_I + will_not_provide_in_full_detail, but_you_can_well_imagine
```

a tool that converted C source into a form suitable for emailing might convert this to one of several forms:

- Splitting long lines is the simplest approach:

```
#line 123
dgay(some_very_long_and_complicated_expression_that_I
                + will_not_provide_in_full_detail,
                but_you_can_well_imagine__that_such_messy_things_ +
                can_easily_arise + in_scientific_computation);
```

  Using line splicing would not help because the method of counting line numbers is not altered by the presence of line splices. 118 line splicing 1986 line number

- Splitting long lines and adding **`#line`** directives so that any diagnostic messages can be related back to the original source might be more developer friendly:

```
#line 123
dgay(some_very_long_and_complicated_expression_that_I
#line 123
         + will_not_provide_in_full_detail,
#line 123
         but_you_can_well_imagine__that_such_messy_things_ +
#line 123
         can_easily_arise + in_scientific_computation);
```

However, if dgay is defined as a macro the behavior will be undefined.

Rationale A new proposal for C99 to allow the **#line** directive to appear within macro invocations was considered. The Committee decided to not allow any preprocessor directives to be recognized as such inside of macros.

predefined macros not #defined 2026

This C specification covers preprocessing directives, not preprocessing operators (such as **defined**, which is discussed elsewhere).

This footnote was added by the response to DR #250.

Common Implementations

As of version 3.3, gcc treats directives within arguments the same as if they had not occurred within macro arguments.

Coding Guidelines

Measurements and experience show that this usage is rare and consequently no guideline recommendations is discussed here.

If there is a **...** in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. 1941

Commentary

This defines the term *variable arguments*. The same term is also used to refer to the arguments corresponding to the ellipsis notation in a function definition. It would be more exact for the specification to say "after the" rather than "in the".

macro replacement

The C preprocessor model of macro expansion is one of performing (potentially recursive) token substitution, not of interpreting sequences of commands (e.g., there is no method of iterating). This model has no existing framework for walking through a list of variable arguments, like statements in a function definition. Without completely rewriting the preprocessor specification there is little scope for anything other than solution adopted by the C Committee. Because all of the variable arguments are formed into a single item they be used in a context that treats them as a single sequence of preprocessing tokens.

C90

Support for **...** in function-like macro definitions is new in C99.

C++

Support for **...** in function-like macro definitions is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

These guideline recommendations are driven by common developer behaviors in dealing with constructs. This construct is new in C99 and as yet no significant experience has been gained about how developers interact with it. The specification of behavior is sufficiently different from the use of ellipsis in function prototype definitions that drawing parallels, with the aim of framing an applicable guideline recommendation, does not look possible.

EXAMPLE 1984 variable macro arguments

The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the **...**). 1942

Commentary

This argument specification differs from that for function definitions, in that for macros at least one argument is required to match the **...** notation. If it is possible that a single argument may be passed then the definition of the macro needs to include **...** as the only parameter in its definition.

```
1  #define ONE(...)
2  #define TWO(p1, ...)
```

The function-like macro ONE requires at least one argument and the macro TWO requires at least two arguments.

1943 146) Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

footnote 146

Commentary

This issue is discussed in more detail elsewhere.

1951 # stringize operand

Other Languages

Some early implementations had preprocessors that were character based and scanned character constants and string literals for macro names to expand.

Example

```
#define X marks the spot

char *str = "Xmas"; /* Not expanded to "marks the spotmas". */
```

1944 147) Despite the name, a non-directive is a preprocessing directive.

footnote 147

Commentary

This footnote was added by the response to DR #250. In the following example `#nonstandard` has the status of a preprocessing directive and the behavior is undefined:

```
#define nothing(x)

nothing (
#nonstandard
)
```

6.10.3.1 Argument substitution

1945 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place.

argument substitution

Commentary

The term *argument substitution* is also commonly used by developers to refer to this process.

How a sequence of preprocessing tokens within a source file are split into the arguments that belong to a particular function-like macro invocation is discussed elsewhere.

1939 macro arguments separated by comma

Coding Guidelines

Experience suggests that many developers do not have accurate knowledge of the sequence of operations that occur during argument substitution. In many cases this lack of knowledge is not significant because the final result is as expected. In more subtle cases the results may be surprising. However, your author is not aware of any pattern to these developer misconceptions and so is unable to word any guideline recommendation. Given that there are few cases where a detailed knowledge of the sequence of operations is needed education may not help (i.e., without practice developers are unlikely to remember what they learned some time ago). Developer uncertainty about the sequence of events may lead them to select the most likely behavior from among a range of possibilities they consider to be plausible in the given context. There are no obvious guideline recommendations covering this pattern of usage.

1939 macro arguments separated by comma

parameter argument macro expanded

A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. 1946

Commentary

That is, parameters occurring in the replacement list are replaced after their corresponding arguments have been macro expanded. The # and ## operator contexts are the only situations where any macros appearing in a parameter's corresponding argument are not considered for replacement. In the following the expanded form of PARAM is not examined for preprocessing tokens that have the same spelling as a parameter of F.

```
1  #define PARAM param
2  #define F(param) param + PARAM
3
4  int glob = F(1); /* Expands to 1+param, rather than 1+1 */
```

Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; 1947

Commentary

Each argument is expanded in isolation (i.e., there is no interaction between arguments or any other preprocessing tokens in the source file).

Example

In the following the argument in the invocation of FM2 is expanded to the two tokens {FM1} and {(}.

```
1  #define L_PAREN (
2  #define FM1(a) a
3  #define FM2(b) b 23)
4
5  void f1(void)
6  {
7  FM2(FM1 L_PAREN);
8  }
```

Completely expanding the argument requires that FM1 then be expanded (it is followed by an opening parentheses). However, this expansion does not succeed because there is no matching closing parentheses, in the sequence of preprocessing tokens for that argument. The behavior is undefined. Continuing on from the above source:

```
1  void f2(void)
2  {
3  FM2(FM1(L_PAREN));
4  }
```

FM2 expands to (23). The following definition achieves what may have been intended:

```
1  #define FM3(c, d) c d 23)
2
3  void f3(void)
4  {
5  FM3(FM1, L_PAREN);
6  }
```

the invocation of FM3 expands to 23 (the argument FM1 is not expanded further, as an argument, because it is not followed by an opening parentheses).

1948 no other preprocessing tokens are available.

Commentary

In particular any subsequent preprocessing tokens from the source file, or any preprocessing tokens following the parameter in the replacement list.

1949 An identifier **__VA_ARGS__** that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

Commentary

This is a requirement on the implementation.

This is replaced by all the arguments that match the ellipsis, including the commas between them. Rationale

The extent to which the arguments corresponding to the parameter **__VA_ARGS__** are replaced may be affected by the presence of commas. For instance, in:

```
#define LPAREN (
#define RPAREN )
#define F(x, y) x + y
#define ELLIP_FUNC(...) __VA_ARGS__

ELLIP_FUNC(F, LPAREN, 'a', 'b', RPAREN); /* 1st invocation */
ELLIP_FUNC(F  LPAREN  'a', 'b'  RPAREN); /* 2nd invocation */
```

the argument in the second invocation of `ELLIP_FUNC` expands to `F('a', 'b')` which in turn expands to `'a'+'b'`. This expansion sequence does not occur in the first invocation because of the comma separating `F` from `(` (and other preprocessing tokens).

C90

Support for **__VA_ARGS__** is new in C99.

C++

Support for **__VA_ARGS__** is new in C99 and is not specified in the C++ Standard.

6.10.3.2 The # operator

Constraints

1950 Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list. # operator

Commentary

Within a replacement list the **#** preprocessing token is a unary operator. It is commonly called the *stringize* operator.

This operator is intended to be applied to the unexpanded form of the corresponding argument. Other preprocessing tokens in the replacement list can be *stringized* by simply delimiting them in double-quotes. Occurrences of a **#** preprocessing token that are not followed by a parameter are probable unintended and this constraint requirement ensures that translators will issue a diagnostic.

Usage

Based on the visible form of the `.c` files 0.26% (0.09% `.h` files) of the replacement lists of macro definitions contained a **#** operator. There were no obvious patterns to the usage.

Semantics

stringize operand

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. 1951

Commentary

Rationale

Some pre-C89 implementations decided to replace identifiers found within a string literal if they matched a macro argument name. The replacement text is a "stringized" form of the actual argument token sequence. This practice appears to be contrary to K&R's definition of preprocessing in terms of token sequences. The C89 Committee declined to elaborate the syntax of string literals to the point where this practice could be condoned; however, since the facility provided by this mechanism seems to be widely used, the C89 Committee introduced a more tractable mechanism of comparable power.

In the following example:

```
#define HASH #

#define M(a) HASH a /* Expands to # a */
```

the # preprocessing token exists in the expanded replacement list, rather than the replacement list and is considered to be a punctuator rather than an operator.

Common Implementations

Microsoft C supports the preprocessor operator `#@` (call the *charizing* operator) as an extension. It converts its operand to a character constant.

Example

```
#define mkstr(a) # a

char *p_1 = mkstr("mnp\a");    /* Assigns "\"mnp\\a\"" */
char *p_2 = mkstr(000);        /* Assigns "000"        */
char *p_3 = mkstr(0);          /* Assigns "0"          */
char *p_4 = mkstr(0004);       /* Assigns "0004"       */
char *p_5 = mkstr(0.001E+000); /* Assigns "0.001E+000" */
char *p_6 = mkstr(x\
yz);                           /* No new-line in created string, i.e., "xyz" */
char *p_7 = mkstr(0
K);                            /* Assigns "0 K" */
```

white space between macro argument tokens

Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. 1952

Commentary

white-space 128 sequence replaced by one

Whether multiple white space between preprocessing tokens has already been converted to a single white space before this conversion is discussed elsewhere. Also there is no white space added where none existed in the source file.

Rationale

One problem with defining the effect of stringizing is the treatment of white space occurring in macro definitions. Where this could be discarded in the past, now upwards of one logical line may have to be retained. As a compromise between token-based and character-based preprocessing disciplines, the C89 Committee decided to permit white space to be retained as one bit of information: none or one. Arbitrary white space is replaced in the string by one space character.

Coding Guidelines

Any misconceptions about the white space will appear between preprocessing tokens that have been stringized is a developer education issue, not a coding guideline issue.

Example

The following assigns the string literal "x y \\a+ 1" to p.

```
#define mkstr(a) # a

char *p = mkstr(x y   \a+
                          1);
```

Your author cannot think of a way to generate multiple, adjacent, space characters in a stringized sequence of preprocessing tokens.

1953 White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted.

Commentary

This specification mirrors the one given for the replacement list and it is given for the same reasons.

1929 white-space before/after replacement list

1954 Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name.

escape sequence handling

Commentary

This specification is intended to ensure that the output produced by passing the string produced by the stringize operator as an argument to `printf`, for instance, is the same as the visible form (with white-space characters reduced to a single space character) of the preprocessing token sequence immediately prior to being stringized (although this sequence may not exist in the visible source). In the following example:

```
#define mkstr(s) #s

char *at = mkstr(\u0040);
```

if UCNs are mapped in translation phase 1 and @ is a supported character then the invocation of `mkstr` expands to "@". Otherwise the conversion occurs in translation phase 5 and the invocation expands to "\\u0040".

116 translation phase 1
133 translation phase 5

C90

Support for universal character names is new in C99.

C++

Support for universal character names is available in C++. However, wording for this clause of the C++ Standard was copied from C90, which did not support universal character names. The behavior of a C++ translator can be viewed as being equivalent to another C99 translator, in this regard. A C++ translator is not required to document its handling of a \ character before a universal character name.

Example

The # operator provides a mechanism for producing defined behavior from what appears to be a sequence of preprocessing tokens having no guaranteed interpretation within the C Standard.

776 character ' or " matches

```
#define mkstr(x) #x

char *p = mkstr(');  /* A single quote might, subject to undefined behavior, be given a meaning. */
```

If the replacement that results is not a valid character string literal, the behavior is undefined. 1955

Commentary

string literal 895 syntax

For instance, syntactically, occurrences of the backslash character in string literals are limited to escape sequences. In the following example:

```
#define mkstr(x) # x

char *p = mkstr(a \ b); /* "a \ b" violates the syntax of string literals */
```

the result of the # operator need not be `"a \ b"`.

Common Implementations

translation phase 133 5

Most implementations simply create a sequence of characters. However, processing in subsequent phases of translation (e.g., conversion of escape sequences) may also result in undefined behavior.

The character string literal corresponding to an empty argument is `""`. 1956

Commentary

This specification is more consistent than the stringize operator not returning any preprocessing token for this case.

C90

An occurrence of an empty argument in C90 caused undefined behavior.

C++

Like C90, the behavior in C++ is not explicitly defined (some implementations e.g., Microsoft C++, do not support empty arguments).

and ## evaluation order

The order of evaluation of # and ## operators is unspecified. 1957

Commentary

The C committee chose not to specify the relative precedence of these operators. In:

```
#define STR_GLUE(a, b) # a ## b

char *p = STR_GLUE(1, 2);
```

if `{1}` is glued to `{2}` and then stringised the resulting preprocessing token is defined. However, stringizing `{1}` and then attempting to glue it to `{2}` does not yield a defined preprocessing token (the behavior is undefined).

When both operators occur in a replacement list, performing token gluing first would appear to give the highest probability of having a defined result, when a stringize operator is also present. However, there is no requirement that implementations use this order. There is no need to specify an evaluation order for the # operator because it is unary (the evaluation order or the ## operator is discussed elsewhere).

1965 evaluation order

Coding Guidelines

full expression 1712

An order dependency on the evaluation of the # and ## operators only exists when either of them could be applied to the same preprocessing token in a replacement list. Unlike full expression evaluation it is not possible to use parentheses to group operands with preprocessor operators. These operators have to be adjacent to the preprocessing tokens they operate on. However, this combination of events rarely occurs (there are no occurrences in the `.c` files) and thus a guideline recommendation is not considered worthwhile.

Example

It is possible to specify an ordering by breaking the operations out into separate macro definitions.

```
#define MK_STR(a) #a
#define GLUE(a, b) a ## b

char *p_1 = MK_STR(GLUE(1, 2));

#define STR_GLUE(a, b) MK_STR(a ## b)

char *p_2 = STR_GLUE(1, 2);
```

6.10.3.3 The ## operator

Constraints

1958 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition. ## operator

Commentary

The ## preprocessing token is a binary operator, requiring two operands. This operator is often called the *token gluing*, *glue*, or *token pasting* operator. Unlike the # operator, there is no requirement that the ## operator only be applied to preprocessing tokens that are parameters.

Common Implementations

gcc supports an additional use for the ## preprocessing operator. If a macro parameter accepting a variable number of arguments is passed no arguments and a ## appears before its use in the replacement list, the preceding comma punctuator is removed from the expanded replacement list. For instance, in:

```
#define eprintf(format, args...)         \
      fprintf (stderr, format , ## args)

/* The invocation */

eprintf("failure!\n");

/* is expanded to:
 *
 *    fprintf(stderr, "failure!\n");
 *
 * rather than to:
 *
 *    fprintf(stderr, "failure!\n" , );
 */
```

The Plan 9 C compiler intentionally lacks support for the ## operator.[1087]

Example

Readers might like to work out which of the following two assignments to max relies on undefined behavior and which is strictly conforming.

file_1.c

```
#define __CONCAT__(_A, _B) _A ## _B
#define __CONCAT_U__(_A) _A ## u
#define ULONG32_C(__c)  __CONCAT__(__CONCAT_U__(__c), l)
#define ULONG32_MAX   ULONG32_C(4294967295)

unsigned long max = ULONG32_MAX;
```

file_2.c
```
#define __XXCONCAT__(_A, _B) _A ## _B
#define __CONCAT__(_A, _B) __XXCONCAT__(_A, _B)
#define __XCONCAT_U__(_A) _A ## u
#define __CONCAT_U__(_A) __XCONCAT_U__(_A)
#define ULONG32_C(__c)  __CONCAT__(__CONCAT_U__(__c), l)
#define ULONG32_MAX   ULONG32_C(4294967295)

unsigned long max = ULONG32_MAX;
```

Table 1958.1: Occurrence of the ## preprocessor operator (as a percentage of all occurrences of that operator). The form , ## identifier is a gcc extension (described in the Common implementations subclause). Based on the visible form of the .c and .h files.

Preprocessing Token Sequence	%
identifier ## identifier	70.2
, ## identifier	24.2
identifier ## identifier ## identifier	15.7
others	4.8
integer-constant ## identifier	1.8
integer-constant ## identifier ## *integer-constant*	1.0
identifier ## *integer-constant*	1.0

Semantics

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; 1959

Commentary

parameter 1946
argument macro expanded

In this case the argument will not have been macro expanded before this replacement occurs (so if it consists of more than one preprocessing token only the first, and/or the last, are operated on).

Rationale

Another facility relied on in much current practice but not specified in K&R is "token pasting," or building a new token by macro argument substitution. One pre-C89 implementation replaced a comment within a macro expansion by no characters instead of the single space called for in K&R . The C89 Committee considered this practice unacceptable.

As with "stringizing," the facility was considered desirable, but not the extant implementation of this facility, so the C89 Committee invented another preprocessing operator. The ## operator within a macro expansion causes concatenation of the tokens on either side of it into a new composite token.

The specification of this pasting operator is based on these principles:

- Paste operations are explicit in the source.
- The ## operator is associative.
- A formal parameter as an operand for ## is not expanded before pasting. The actual parameter is substituted for the formal parameter; but the actual parameter is not replaced. Given, for example

  ```
  #define a(n) aaa ## n
  #define b    2
  ```

  the expansion of a(b) is aaab, not aaa2 or aaan.
- A normal operand for ## is not expanded before pasting.

- Pasting does not cross macro replacement boundaries.
- The token resulting from a paste operation is subject to further macro expansion.

These principles codify the essential features of prior art and are consistent with the specification of the stringizing operator.

Example

```
#define GLUE(a, b) a ## b

double d = GLUE(3.4e, -3); /* surprise! -3 is two preprocessing tokens */
```

1960 however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[148)]

argument no tokens replaced by placemarker token

Commentary

This defines the term *placemarker*. The standard uses placemarker preprocessing tokens to describe an effect, an implementation need not represent them internally. The need for a placemarker preprocessing token occurs because the **##** operator does not cross replacement boundaries.

C90

The explicitly using the concept of a placemarker preprocessing token is new in C99.

C++

The explicit concept of a placemarker preprocessing token is new in C99 and is not described in C++.

Coding Guidelines

In C90, an argument that consisted of no preprocessing tokens resulted in undefined behavior. The extent to which developers made use of such arguments and the behavior they expected from such usage is not known. Whether the definition of a behavior, in C99, and the introduction of the placemarker concept is something that developers need to be made aware of is not known.

1961 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token.

Commentary

The preprocessing token **##** is only recognized as an operator when it appears in the replacement list of the macro definition. When it appears anywhere else it is a punctuator.

1966 EXAMPLE # ## #
1980 EXAMPLE reexamination
1981 EXAMPLE # and ##

Table 1961.1: Possible results of concatenating, using the **##** operator, pairs of preprocessing tokens (the one appearing in the left column followed by the one appearing in the top row) where the result might be defined (*undefined* denotes undefined behavior).

	identifier	*pp-number*	*punctuator*	*string-literal*	*character-constant*
identifier	identifier	identifier or undefined	undefined	string-literal or undefined	character-constant or undefined
pp-number	pp-number	pp-number	pp-number or undefined	undefined	undefined
punctuator	pp-number or undefined	pp-number or undefined	punctuator or undefined	undefined	undefined
everything else	undefined	undefined	undefined	undefined	undefined

placemarker preprocessor

Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. 1962

Commentary

EXAMPLE 1982 placemarker

This specification handles the special case of an argument containing no preprocessing tokens.

C90

The concept of placemarker preprocessing tokens is new in the C99 Standard. The behavior of concatenating an empty argument with preprocessing token was not explicitly defined in C90, it was undefined behavior.

C++

Like C90, the behavior of concatenating an empty argument with preprocessing token is not explicitly defined in C++, it is undefined behavior.

Example

N418.c

```
#define PI 3.1416
#define  F f
#define  D     /* Expands into no preprocessing tokens. */
#define LD L
#define  glue(a, b) a ## b
#define xglue(a, b) glue(a, b)

/*
 * The following:
 */
float        f = xglue(PI, F);
double       d = xglue(PI, D);
long double ld = xglue(PI, LD);
/*
 * should expand into:
 */
float        f = 3.1416f;
double       d = 3.1416;
long double ld = 3.1416L;
```

if result not valid

If the result is not a valid preprocessing token, the behavior is undefined. 1963

Commentary

translation phase 136, preprocessing token 7 771 shall have lexical form

While behavior in later phases is also likely to be undefined (e.g., when the preprocessing token is converted to a token) and a constraint violation, this specification applies during preprocessing and removes the need to define the behavior of any subsequent operations involving the result.

Common Implementations

Many implementations allow invalid preprocessing tokens to be created and to subsequently be operands of the ## operator. This behavior enables the creation of preprocessing tokens that need to be built out of three separate preprocessing tokens, where the result of concatenating any two of them is not a valid preprocessing token.

Coding Guidelines

This situation is not sufficiently commonly for a guideline recommendation to be worthwhile.

Example

```
#define GLUE(a, b, c) a ## b ## c

extern void f(int p, GLUE(. , . , .));

#include GLUE(< , header, >)
```

1964 The resulting token is available for further macro replacement.

Commentary

However, it is not available for use as a preprocessing operator. The result of concatenating two preprocessing tokens is not treated any differently than other preprocessing tokens. 1966 EXAMPLE # ### #

1965 The order of evaluation of ## operators is unspecified. ## evaluation order

Commentary

If all intermediate results are defined for all orders of evaluation, the final result will always be the same. However, in some cases the result is not defined for some orders of evaluation.

Coding Guidelines

It is not possible to use parentheses to define an order of evaluation (the operators have to be adjacent to the preprocessing tokens that they operate on). The only solution is to break the replacement list up into separate macro definitions, each performing a single operation.

Example

```
#define GLUE_3(x, y, z) x ## y ## z

GLUE_3(>, >, =)    /* Behavior defined for all orders of evaluation.         */
GLUE_3(1, . , e10) /* Behavior only defined if left most ## performed first. */
/* There are no cases where the behavior is only defined if right most ## performed first. */
```

1966 EXAMPLE In the following fragment: EXAMPLE # ## #

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y"
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding **hash_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator.

Commentary

This example was created by the response to DR #017q22.

C++

This example is the response to a DR against C90. While there has been no such DR in C++, it is to be expected that WG21 would provide the same response.

---

footnote 148

148) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4. 1967

Commentary

function-like macro followed by (1935

The standard does not specify any formal syntax for function-like macro invocations, let alone placemarker preprocessing tokens.

C90

Support for the concept of placemarker preprocessing tokens is new in C99.

C++

Support for the concept of placemarker preprocessing tokens is new in C99 and they are not described in the C++ Standard.

6.10.3.4 Rescanning and further replacement

---

rescanning

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemarker preprocessing tokens are removed. 1968

Commentary

The concept of placemarkers is only needed during the processing of the **##** operator.

C90

Support for the concept of placemarker preprocessing tokens is new in C99.

C++

Support for the concept of placemarker preprocessing tokens is new in C99 and does not exist in C++.

---

rescanned along with subsequent tokens

Then, the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace. 1969

Commentary

Unlike argument substitution, and **#** and **##** operator processing, the replacement list is not processed in isolation from the rest of the source code. For instance, in:

```
#define M1(a) (a+1)
#define M2(b) b

int ei_1 = M2(M1)(17);   /* becomes int ei_1 = (17+1);   */
int ei_2 = (M2(M1))(17); /* becomes int ei_2 = (M1)(17); */
```

after the invocation of `M2` is expanded, the resulting sequence is `M1`. The rest of the source code is `(17);`. Rescanning including this sequence of preprocessing tokens, in the assignment to `ei_1`, results in an invocation of the macro `M1`.

Rationale

The rescanning rules incorporate an ambiguity. Given the definitions

```
#define  f(a)  a*g
#define  g     f
```

it is clear (or at least unambiguous) that the expansion of `f(2)(9)` is `2*f(9)`, the `f` in the result being introduced during the expansion of the original `f`, and so is not further expanded.

However, given the definitions

```
#define f(a)  a*g
#define g(a)  f(a)
```

the expansion will to be either `2*f(9)` or `2*9*g`: there are no clear grounds for making a decision whether the `f(9)` token string resulting from the initial expansion of `f` and the examination of the rest of the source file should be considered as nested within the expansion of `f` or not. The C89 Committee intentionally left this behavior ambiguous as it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.

Coding Guidelines

There are situations where it is intended that a replacement list include preprocessing tokens from the rest of the source file. For instance, if the name of a function needs to be unconditionally mapped to another name an object-like macro needs to be used. However, after expansion the mapped name might invoke a function-like macro:

```
1  #define isprint __PRINT_PROPERTY
2  #define __PRINT_PROPERTY(x) (((x < 0) || (x > 127)) ? 0 : __IS_PRINT[x])
3  int (__PRINT_PROPERTY)(int);
4
5  void f(void)
6  {
7  _Bool a_printable = isprint('a');
8  }
```

1970 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced.

macro being replaced found during rescan

Commentary

A problem faced by many pre-C89 preprocessors is how to use a macro name in its expansion without suffering "recursive death." The C89 Committee agreed simply to turn off the definition of a macro for the duration of the expansion of that macro. Rationale

Coding Guidelines

Whether or not a macro expansion that depends on this recursion-breaking rule requires significantly greater effort to comprehend than other macro expansions, involving similar numbers of preprocessing tokens but no recursion breaking, is not known. Neither is it known whether an alternative set of macros, that did not depend on recursion breaking, would require less effort. Without this information it is not possible to estimate the cost/benefit of any guideline recommendations and none are made here.

Example

```
1  extern int M_1,
2             M_2;
```

```
3
4   #define M_1 M_2
5   #define M_2 M_1
6
7   void f(void)
8   {
9   M_1 = M_2; /* Macros do not alter the behavior, i.e., M_2 is still assigned to M_1 */
10  }
```

```
1   #define short static short
2
3   short si; /* Expands to static short si; */
```

EXAMPLE 1980 reexamination Also see elsewhere for examples.

Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. 1971

Commentary

Indirect recursion, via other macro definitions, is treated the same as direct recursion.

Coding Guidelines

The reasoning here is the same as for the case of direct recursion.

Example

```
1   static int M_0 = 0;
2
3   #define M_0(x)     M_ ## x
4   #define M_1(x) x + M_0(0)
5   #define M_2(x) x + M_1(1)
6   #define M_3(x) x + M_2(2)
7   #define M_4(x) x + M_3(3)
8   #define M_5(x) x + M_4(4)
9
10  int f_1(void)
11  {
12  return M_0(1)(2)(3)(4)(5); /* Expands to:
13                                          2 + M_0(3)(4)(5)
14                                              or
15                                          2 + M_0(0)(3)(4)(5) */
16  }
17
18  int f_2(void)
19  {
20  return M_0(5)(4)(3)(2)(1); /* Expands to: 4 + 4 + 3 + 2 + 1 + M_0(3)(2)(1) */
21  }
```

These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced. 1972

Commentary

The C preprocessor model is one where the incoming preprocessing tokens are processed and then output. It is not intended that the preprocessor have to hold on to all preprocessing tokens it has processed, until the end of the source file is reached.

```
#define F(a) a
#define FUNC(a) (a+1)

void f(void)
{
/*
 * The preprocessor works successively through the input without
 * backing up through previous processed preprocessing tokens.
 */
F(FUNC) FUNC (3); /* final token sequence is FUNC(3+1) */
}
```

This rule allows implementations to mark preprocessing tokens with a single bit (this bit is often referred to using the term *blue paint*, after the marking ink used by engineers, by members of the C committee), indicating that they are no longer available for replacement.

Example

```
#define A   A B C
#define B   B C A
#define C   C A B

A
/*
 * Using the notation:
 *    X={ }       the result of expanding X.
 *    lowercase   an identifier that has been 'painted blue'.
 * 'simplify'
   expand    A={ A B C }
   paint     A={ a B C }
   rescan    A={ a B={ B C A } C }
   paint     A={ a B={ b C a } C }
   rescan    A={ a B={ b C={ C A B } a } C }
   paint     A={ a B={ b C={ c a b } a } C }
             A={ a B={ b c a b a } C }
             A={ a b c a b a C }
   rescan    A={ a b c a b a C={ C A B }}
   paint     A={ a b c a b a C={ c a B }}
   rescan    A={ a b c a b a C={ c a B={ B C A }}}
   paint     A={ a b c a b a C={ c a B={ b c a }}}
             A={ a b c a b a C={ c a b c a }}
             A={ a b c a b a c a b c a }

   simplify  a b c a b a c a b c a

 * Final tokens output:

   A B C A B A C A B C A
 */
```

1973 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

expanded token sequence not treated as a directive

Commentary

As described elsewhere, a preprocessing directive is a particular sequence of preprocessing tokens at the start of translation phase 4. Macro-replaced preprocessing tokens don't exist until after the start of translation

1855 preprocessing directive consists of

argument 1940 resemble preprocessing directive
_Pragma 2030 operator

phase 4. The issue of arguments that resemble preprocessing directives is discussed elsewhere. The **_Pragma** unary operator is discussed elsewhere.

C90

Support for **_Pragma** unary operator expressions is new in C99.

C++

Support for **_Pragma** unary operator expressions is new in C99 and is not available in C++.

Example

```
#define H #
#define D define

#define DEFINE(a, b) H D a b

DEFINE(X, 3)
```

the invocation results in the sequence of preprocessing tokens:

{#} {define} {X} {3}

which are not treated as a preprocessing directive.

6.10.3.5 Scope of macro definitions

---

macro definition lasts until

A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. 1974

Commentary

There is no requirement that macros with the same names in different translation units have the same replacement lists, or both be object-like or function-like (neither linkage or scope apply to macro names).

linkage 420 scope 390 of identifiers

Common Implementations

Very few implementations write information on macro definitions out to the generated object file. Although some static analysis tools save this information for later cross translation unit consistency checking.

Coding Guidelines

The issue of having the definition of macro names exist over some kind of restricted scope is discussed elsewhere.

macro definition 408 emulate block scope

---

macro definition no significance after

Macro definitions have no significance after translation phase 4. 1975

Commentary

All preprocessing directives are deleted at the end of translation phase 4.

preprocessing directives 132 deleted

C90

This observation is new in C99.

C++

This observation is not made in the C++ document.

#undef

A preprocessing directive of the form 1976

```
# undef identifier new-line
```

causes the specified identifier no longer to be defined as a macro name.

Commentary

The relatively high percentage of macro definitions that do not include a replacement list (see Table 1931.1) shows the degree to which the status of being defined as a macro name is sufficient information for developer use. The **#undef** directive provides additional control over which identifiers are defined as macro names.

The standard specifies a few identifiers that cannot be **#undef**ed.

2026 predefined macros not #defined

Common Implementations

Some translators support a `-U` translator option that can be used to override any predefined macros (either defined using the `-D` option or internally generated by the translator). Some prestandard translators handled **#define**/**#undef** in a stack-like fashion.

1919 #define/#undef stack

Coding Guidelines

Discussion of the **#undef** directive, in guideline documents or between developers, is relatively rare. Whether this is because use of this directive rare, or simply innocuous is not known.

Usage

Approximate 5% of all **#undef** directives occur before a **#include** directive (based on the visible form of the `.c` files).

Table 1976.1: Occurrence of various sequences of preprocessing directives (as a percentage of all such sequences) that follow a **#undef** and reference the same identifier (e.g., 2.7% of the first occurrence of **#undef** are followed by one or more **#define**s followed by one or more **#undef**s). **#define** represents one or more **#define** preprocessing directives. **#undef** represents one or more **#undef** preprocessing directives. **#if[n]def** represents two or more **#if**s and **#ifndef**s, in any order. **#und-def** represents one or more pairs of **#undef** **#define** preprocessing directives. Based on the visible form of the `.c` files.

Following Directive Sequences	%
	53.0
#ifdef	20.4
#define	16.2
others	4.8
#define #undef	2.7
#if(n)def	1.5
#define #undef-#define **#undef**	1.4

1977 It is ignored if the specified identifier is not currently defined as a macro name.

Commentary

This behavior simplifies the process of using **#undef** by removing the need to check the status of the identifier (e.g., by using **#ifdef**) before evaluating the directive.

Coding Guidelines

The issue of redundant code is discussed elsewhere. Whether or not a **#undef** is redundant, or simply providing a fail safe backup is outside the scope of these guidelines.

190 redundant code

1978 EXAMPLE 1 The simplest use of this facility is to define a “manifest constant”, as in

```
#define TABSIZE 100
int table[TABSIZE];
```

Commentary

Defining an object-like macro to have an integer constant replacement list is one of the most commonly given examples of the use of the preprocessor.

EXAMPLE macro argument side effects

EXAMPLE 2 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. Also it cannot have its address taken, as it has none. 1979

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

Commentary

macro 1939 arguments separated by comma

The issue of side effects in the evaluation of macro arguments is discussed elsewhere.

EXAMPLE reexamination

EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence 1980

```
#define x      3
#define f(a)   f(x * (a))
#undef  x
#define x      2
#define g      f
#define z      z[0]
#define h      g(~
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
       (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5,  };
char c[2][6] = { "hello" "" };
```

Commentary

Analyzing a part of the example, we get:

```
1  #define f(a)   f(x * (a))
2  #define x      2
3  #define z      z[0]
4
5  f(f(z));
```

the sequence of expansions are (preprocessing tokens bracketed by {} are no longer available for further replacement):

```
1  f(f(z))
```

replacing the argument z:

```
1  f(f({z}[]))
```

substituting and expanding the nested function-like macro:

```
f({f}(2 * ({z}[0])))
```

the argument has been fully expanded and can now be substituted into the replacement list:

```
f(2 * ({f}(2 * ({z}[0]))))
```

and expanding the replacement list we get:

```
{f}(2 * ({f}(2 * ({z}[0]))))
```

so the final sequence of preprocessing tokens is:

```
f(2 * (f(2 * (z[0]))))
```

1981 EXAMPLE 4 To illustrate the rules for creating character string literals and concatenating tokens, the sequence EXAMPLE # and ##

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s" \
                        x ## s, x ## t)
#define INCFILE(n)  vers ## n
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d" "abc", '\4') // this goes away
      == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

Commentary

The characters "(after macro replacement, before file access)" are commentary and are not generated by macro replacement.

EXAMPLE placemarker

EXAMPLE 5 To illustrate the rules for placemarker preprocessing tokens, the sequence 1982

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
            10, 11, 12,  };
```

Commentary

1965 evaluation order

Although the order of evaluation of the **##** operator is unspecified, in the above example all orders return the same result.

C90

This example is new in the C99 Standard and contains undefined behavior in C90.

C++

The C++ Standard specification is the same as that in the C90 Standard,

EXAMPLE macro redefinition

EXAMPLE 6 To demonstrate the redefinition rules, the following sequence is valid. 1983

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)   ( a )
#define FUNC_LIKE( a )(        /* note the white space */ \
                         a /* other stuff on this line
                            */ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)     // different token sequence
#define OBJ_LIKE    (1 - 1) // different white space
#define FUNC_LIKE(b) ( a )  // different parameter usage
#define FUNC_LIKE(b) ( b )  // different parameter spelling
```

Commentary

The preprocessor is not required to have any knowledge of how subsequent phrases of translation treated sequences of preprocessing tokens (i.e., the fact that they become an integer constant expression that has the same value as the replacement list of another macro definition).

EXAMPLE variable macro arguments

EXAMPLE 7 Finally, to show the variable argument list macro facilities: 1984

```
#define debug(...)        fprintf(stderr, __VA_ARGS__)
#define showlist(...)        puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr,  "Flag" );
fprintf(stderr,  "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
                printf("x is %d but y is %d", x, y));
```

C90

Support for macros taking a variable number of arguments is new in C99.

C++

Support for macros taking a variable number of arguments is new in C99 and is not supported in C++.

6.10.4 Line control

Constraints

1985 The string literal of a **#line** directive, if present, shall be a character string literal. #line

Commentary

This requirement that character string literals be used (i.e., no wide string literals) is more restrictive than that for the **#include** directive (which specifies implementation-defined handling of the sequence of characters). 1899 #include q-char-sequence

Semantics

1986 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token. line number

Commentary

This defines the term *line number*. Counting new-line characters read or introduced in translation phase 1 means that line splicing does not affect the line number. The first line of the source file has a line number of 1. 116 translation phase 1

1987 A preprocessing directive of the form #line digit-sequence

```
# line digit-sequence new-line
```

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer).

Commentary

Developers point of reference for diagnostic messages is invariably the contents of the untranslated source file. The **#line** directive can simplify the implementations of tools that modify this source during translation, such as a preprocessor. Using this directive removes the need for them (in those cases where a mapping back to the original line number is desirable) to ensure that modifications to the source maintain line number information.

Common Implementations

Some implementations supported the following as an equivalent form:

```
# digit-sequence new-line
```

In some cases this token sequence is generated by translation phase 4 to provide line number information to subsequent phases of translation (to enable them to provide accurate line number information in any diagnostics issued).

1988 The digit sequence shall not specify zero, nor a number greater than 2147483647.

Commentary

The syntax of the directive does not permit a minus sign to appear before the digits. 1854 preprocessor directives syntax

C90

The limit specified in the C90 Standard was 32767.

C++

Like C90, the limit specified in the C++ Standard is 32767.

Common Implementations

Many existing C90, and C++, implementations supported the C99 value.

#line
digit-sequence
s-char-sequence

A preprocessing directive of the form 1989

```
# line digit-sequence "s-char-sequence_opt" new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

Commentary

Hiding the implementation details about the name of the file containing generated, or preprocessed, source can have a variety of advantages (e.g., the contents of a **#include**d file may refer to the name of the source file from which it was generated). This option provides such functionality.

Common Implementations

Some early implementations supported the following as an equivalent form:

```
# digit-sequence "s-char-sequence_opt" new-line
```

A preprocessing directive of the form 1990

```
# line pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted.

Commentary

This form provides some flexibility in allowing the line number and source filename to be specified via macro replacement.

#line
macros expanded

The preprocessing tokens after **line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). 1991

Commentary

To be exact, the preprocessing tokens after **line** on the directive up to the first new-line character are processed just as in normal text.

The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate. 1992

Commentary

However, preprocessing tokens having the form of a constant expression are not evaluated to create an integer constant. For instance, the following will not cause the current line number to be incremented by 10.

```
1  #line __LINE__+10
```

C++

The C++ Standard uses different wording that has the same meaning.

16.4p5

If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

6.10.5 Error directive

Semantics

1993 A preprocessing directive of the form #error

```
# error pp-tokens_opt new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

Commentary

The **#error** directive was introduced in C89 to provide an explicit mechanism for forcing translation to fail under certain conditions. Formally, the Standard can require only that a diagnostic be issued when the **#error** directive is processed. It is the intent of the Committee, however, that translation cease immediately upon encountering this directive if this is feasible in the implementation. Further diagnostics on text beyond the directive are apt to be of little value. Rationale

C++

. . . , and renders the program ill-formed. 16.5p1

Both language standards require that a diagnostic be issued. But the C Standard does not specify that the construct alters the conformance status of the translation unit. However, given that the occurrence of this directive causes translation to terminate, this is a moot point. 89 #error terminate translation

Common Implementations

Most implementation issue the diagnostic and then stop translation. However, a few implementations do continue translating after this directive is encountered. Some implementations also support the preprocessing directives **#inform**[353] and **#warning**.[1278] As their identifier name suggests they generate informational messages and warnings respectively. The translation does not fail. #warning

Example

```
1  #include <limits.h>
2
3  #if CHAR_BIT != 9
4  #error This program only runs on a host where CHAR_BIT == 9
5  #endif
```

6.10.6 Pragma directive

Semantics

#pragma directive

A preprocessing directive of the form 1994

 # pragma *pp-tokens*$_{opt}$ *new-line*

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)[149] causes the implementation to behave in an implementation-defined manner.

Commentary

Rationale The **#pragma** directive was added in C89 as the universal method for extending the space of directives.

The word *pragma* is from the Greek word meaning *action*. The term *compiler directive* or simply *directive* is often used to refer to **pragma** directives.

In the following example the preprocessing token **STDC** does not immediately follow **pragma** (prior to macro replacement) and implementation-defined behavior can occur.

```
#define GLOBAL_FP_CONTRACT STDC FP_CONTRACT ON

#pragma GLOBAL_FP_CONTRACT
```

C90

The exception for the preprocessing token **STDC** is new in C99.

C++

The exception for the preprocessing token **STDC** is new in C99 and is not specified in C++.

Other Languages

A common implementation mechanism, across all languages, is for certain sequences of characters within a comment to be given a special meaning. A few languages (e.g., Ada and Algol 68) have pragma directives defined in their specification. A larger number of language implementations provide some form of **pragma** directive (although the identifier used to denote it varies). A study by Fowler[439] examined implementation-dependent pragmas in Ada compilers. He found that:

1. fewer than 5% of the implementations support all the language-defined pragmas,
2. fewer than 30% of the implementations documented their support for language-defined pragmas,
3. fewer than 45% of the implementations gave more than a perfunctory description of implementation-defined pragmas and attributes.

Common Implementations

Many implementations include support for one or more **pragma** directive, some have extensive support.[614, 1314] The commercial pressure on vendors is to support the **pragma** directives specified by the market leader in their niche. Some implementations support a preprocessing directive of the form *#pack pp-tokens new-line*. It is used to specify information about how storage allocation, for various types, is to be performed. An equivalent form using a **pragma** directive might be `#pragma pack(2)`. The OpenMP API[42] for shared-memory parallelism in C makes extensive use of **#pragma**.

It is possible for the implementation-defined behavior to replace the **pragma** directive with an executable statement. For instance:

```
if (cond)
   {
#pragma vendor_X_library_call
   }
```

might expand to (when using a preprocessor that recognizes the **pragma** directive):

```
if (cond)
   {
__X_go_faster_stripes();
   }
```

or to (when using a preprocessor that does not recognize the **pragma** directive):

```
if (cond)
   {
   }
```

Some implementations and tools embed directives in comments. For instance, Lint[672] uses the form `/*UPPERCASEKEYWORD*/`, while Splint[400] uses the form `/*@information@*/`.

Coding Guidelines

The **pragma** directive is one of several possible methods that are usually available to a developer to influence how a translator performs its job (another one is command line options). However, one difference that usually exists between **pragma** directives and other methods is that the former might be applied to part of a source file, rather than all of it. For instance, the second **pragma** in a source file may change the behavior switched on by the first directive. There are a number of potential coding guideline issues associated with **#pragma** directives, including:

- The rationale for the appearance of a **pragma** directive, in source code, is to change a translator's default behavior. Any occurrences of this directive thus require readers to remember and take into account special case information, that differs from their knowledge of the default behavior, about the constructs affected by the directive. The amount of information they will need to remember will depend on whether the directive applies to all or parts of a source file.
- The visibility of the directive to readers. Other methods of altering translator behavior may also be visibly opaque (e.g., command line options in make-files). However, readers are generally aware of the places to look for translator configuration information and while readers may look at the first few lines of a source file for translator options they tend not to look at other places in a source file.
- Cutting and pasting is a relatively common method of editing source. One of the dangers of this technique is that **#pragma** directives may be unintentionally copied.

On method of minimizing many of these problems is to place all **pragma** directives near the start of source files. However, this rather defeats the flexibility offered by this directive in being selectively applied to parts of a source file. If directives apply to complete declarations it may be worthwhile placing those declarations in a separate source file (the issues involved in deciding which declarations to put in which source file are discussed elsewhere).

1810 declarations in which source file

There is no obvious, worthwhile, guideline recommendation that can be made about how to structure the use of **pragma** directives and so nothing more is said about the issue here.

1995 The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner.

Commentary

The definition of implementation-defined behavior does not include these possibilities. The permissive behaviors of the **pragma** directive are thus all those permitted by the definition of implementation-defined behavior plus those listed in this C sentence.

42 implementation-defined behavior

C90

These possibilities were not explicitly specified in the C90 Standard.

C++

These possibilities are not explicitly specified in the C++ Standard.

---

pragma unrecognized ignored

Any such **pragma** that is not recognized by the implementation is ignored. 1996

Commentary

The standard does not delimit the extent to which a pragma needs to be "not recognized" by an implementation (such questions are invariably regarded as being quality-of-implementation issues that are outside the scope of the standard).

```
1   #pragma DING DUNG /* Only valid second pp-token is DONG */
```

Other Languages

Ada specifies similar behavior.

Coding Guidelines

redundant code 190

A **pragma** that is not recognized by the implementation is redundant code. It may not be recognized for a number of reasons, including the following:

- It is not support by the implementation. This usage might be considered harmless.
- The developer has misspelled a directive that is supported by the implementation. This is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines 0 not faults

The extent to which it is cost effective for the author to provide information about the likelihood of a particular **pragma** being supported by various implementations (by, for instance, using a comment or conditional inclusion) is difficult to estimate for the general case. At the time the **pragma** directive is written its author will be aware of the use of implementation-defined behavior but may not have any idea of what other implementations, if any, the source will be ported to (e.g., writers of open source often give no thought to the possibility that a translator other than `gcc` might be used). For these reasons no guideline recommendations are considered here.

If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[150)] whose meanings are described elsewhere: 1997

```
#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
on-off-switch:  one of
                ON        OFF        DEFAULT
```

Commentary

The identifier **STDC** is not reserved in other contexts and a source file may define a macro name that has this spelling. The term *standard pragmas* is commonly used by members of the C committee to refer to these **pragma** directives (it also appears in a footnote).

footnote 150 2002

The behavior of these **pragma** directives are specified in the library section.

The purpose of these pragmas is to inform the implementation that during translation code appearing after an occurrence of one of them is to be treated in some special way (e.g., certain floating-point optimizations can/cannot be performed). In many implementations the floating-point environment is dynamic (i.e., it can be reconfigured during program execution). The effect of any instances of these pragmas is static (i.e., it only has any effect during translation) and it is the developers responsibility to ensure that the dynamic behavior matches that specified statically.

C90

Support for the preprocessing token **STDC** in **pragma** directives is new in C99.

C++

Support for the preprocessing token **STDC** in **pragma** directives is new in C99 and is not specified in the C++ Standard.

Other Languages

Although Ada defines 38 standard pragmas it does not use a prefix to introduce them.

Common Implementations

Some vendors and other standards specify that a specific identifier follow **#pragma**. For instance, the IBM C compiler for AIX[618] uses the identifier **ibm** and the OpenMP[42] uses **omp**.

Some processors only provide static control of floating-point modes, i.e., the rounding direction must be encoded as part of the bit pattern of a generated machine code instruction.

1998 **Forward references:** the **FP_CONTRACT** pragma (7.12.2), the **FENV_ACCESS** pragma (7.6.1), the **CX_LIMITED_RANGE** pragma (7.3.4).

1999 149) An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). footnote 149

Commentary

Specifying that macro replacement does not occur within standard pragmas gives the implementation the freedom to use them within the headers it supplies. A macro name, with the same spelling as a preprocessing token occurring in a standard pragma, defined before a header is included will not affect a translator's interpretation of that pragma.

C90

This footnote is new in C99.

C++

This footnote is new in C99 and is not specified in the C++ Standard.

Common Implementations

This behavior is another aspect of the implementation-defined nature of **pragma** directives. If the market leaders perform macro replacement then it is very likely that other vendors will also perform it.

Those preprocessors that support macro replacement within **pragma** directives only need to examine the preprocessing token following a `#pragma` to know whether or not to perform macro replacement.

Coding Guidelines

1996 pragma unrecognized ignored

Given that unrecognized **pragma** directives are ignored by translators it may take developers some time before they notice a difference in implementation behavior (between the cases when macro replacement occurs and does not occur).

Example

The following example illustrates one technique for effectively performing macro replacement in **pragma** directives:

```
#define wanted ON
#define not_wanted OFF

#define Pragma(...) _Pragma(#__VA_ARGS__)
#define xPragma(...) Pragma(__VA_ARGS__)

xPragma(STDC FP_CONTRACT wanted)
xPragma(STDC FENV_ACCESS not_wanted)
```

If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; 2000

Commentary

The standard only guarantees a standard **pragma** directive to be treated as such is when the specified sequence of preprocessing tokens appears in the source.

C90

Support for standard pragmas is new in C99.

C++

Support for standard pragmas is new in C99 and is not specified in the C++ Standard.

Example

```
#if HAS_FAST_FLOAT_POINT_UNIT
#define DO_CONTRACT STDC FP_CONTRACT ON
#else
#define DO_CONTRACT NULL_PRAGMA
#endif

#pragma DO_CONTRACT
```

an implementation is permitted to behave as if it were the standard pragma, but is not required to. 2001

Commentary

This combination of behavior is subsumed within the definition of implementation-defined behavior and is specified here to clarify that it is permitted in a conforming program.

footnote 150

150) See "future language directions" (6.11.8). 2002

6.10.7 Null directive

Semantics

A preprocessing directive of the form 2003

```
# new-line
```

has no effect.

Commentary

Rationale The existing practice of using empty # lines for spacing is supported in the Standard.

null statement syntax 1731

The preprocessor equivalent of the null statement.

Coding Guidelines

The null directive differs from am empty line in that it contains a visible character. This character may be used to held maintain a distinctive visible edge to any indented directives in the source.

6.10.8 Predefined macro names

2004 The following macro names[151)] shall be defined by the implementation:

macro name predefined

Commentary

This is a requirement on the implementation. These macro names are commonly referred to as the *predefined macros*. There is no clause header (i.e., Description, Semantics, Constraints) given for the requirements in this clause.

Some of the macros specified in the following C sentences expand to string literals. These string literals do not have any special properties associated with them (e.g., there is no requirement that any of them share the same storage).

908 string literal distinct array

Other Languages

Many languages (e.g., Ada and Fortran) support a large number of predefined identifiers (which may be intrinsic or part of a predefined library) supplying a variety of information.

Common Implementations

Most implementations also define macros, internally within the translator (i.e., not in headers), that provide information on the identity of the translator (e.g., a representation of the vendor name) and the host environment (particularly those vendors that supply implementations on a number of environment and host processors).

Table 2004.1: Occurrence of predefined macro names (as a percentage of all predefined macro names; a total of 1,826). Based on the visible form of the `.c` and `.h` files.

Predefined Macro	.c files	.h files	Predefined Macro	.c files	.h files	Predefined Macro	.c files	.h files
__LINE__	42.17	43.47	__TIME__	2.52	0.00	__STDC_IEC_559__	0.00	0.00
__FILE__	36.31	37.77	__STDC_VERSION__	0.00	0.00	__STDC_HOSTED__	0.00	0.00
__STDC__	15.77	18.11	__STDC_ISO_10646__	0.00	0.00			
__DATE__	3.23	0.65	__STDC_IEC_559_COMPLEX__	0.00	0.00			

2005 **__DATE__** The date of translation of the preprocessing translation unit: a character string literal of the form **"Mmm dd yyyy"** , where the names of the months are the same as those generated by the **asctime** function, and the first character of **dd** is a space character if the value is less than 10.

__DATE__ macro

Commentary

129 translation phase 4

This string literal specifies the date when translation phase 4 executed, not when later phases of translation were executed (e.g., the output of this phase may be written to a file for subsequent processing on another day). A change of date during the translation process does not affect the character sequence contained in the string literal. The standard does not specify any accuracy requirements on the value returned by the __DATE__ macro.

2025 predefined macros constant values during translation

Specifying that a space character is to be used if the day value is less than 10 the standard guarantees that the string literal always has the same width.

The discussion given for the __TIME__ macro is applicable here.

Common Implementations

Many translators unconditionally create the value to be used as the result of the __DATE__ macro once, when the translator is first started. This value is then used for all occurrences of __DATE__ during that translation. Translators generally rely on the accuracy of the date available in the environment in which they execute. Microsoft C supports the __TIMESTAMP__ macro as an extension. It expands to a string literal containing the date and time of the last modification of the current source file.

A single call to the `time` library function returns all of the information needed to create the values of the __TIME__ and __DATE__ macros.

__DATE__ date not available

If the date of translation is not available, an implementation-defined valid date shall be supplied. 2006

Commentary

The C Standard cannot require that the environment in which a translator executes be capable of knowing the current date. However, it can require that a value always be supplied by the translator.

Common Implementations

The `time` library function, assuming this is how the translator obtains the current date, returns the value `(time_t)-1` if the current calendar time is not available.

__FILE__ macro

__FILE__ The presumed name of the current source file (a character string literal).[152)] 2007

Commentary

It is only the presumed name because the value may have previously been changed, earlier in the source file being translated, using a **#line** directive, or may have to be guessed (e.g., if the source was read from standard input). #line 1985

Other Languages

Perl supports both the special variable `$PROGRAM_NAME` and the global special constant `__FILE__`.

__LINE__ macro

__LINE__ The presumed line number (within the current source file) of the current source line (an integer constant).[152)] 2008

Commentary

It is only the presumed name because the value may have previously been changed, earlier in the source file being translated, using a **#line** directive. #line 1985

Other Languages

Perl supports the global special constant `__LINE__`.

__STDC__ macro

__STDC__ The integer constant **1**, intended to indicate a conforming implementation. 2009

Commentary

Rationale

The macro `__STDC__` allows for conditional translation on whether the translator claims to be standard-conforming. It is defined as having the value 1. Future versions of the Standard could define it as 2, 3, etc., to allow for conditional compilation on which version of the Standard a translator conforms to. The C89 Committee felt that this macro would be of use in moving to a conforming implementation.

C++

16.8p1 *Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.*

It is to be expected that a C++ translator will not define the **__STDC__**, the two languages are different, although a conforming C++ translator may often behave in a fashion expected of a conforming C translator. Some C++ translators have a switch that causes them to operate in a C compatibility mode (in this case it is to be expected that this macro will be defined as per the requirements of the C Standard).

Common Implementations

Some implementations define this macro to have the value 0 when operating in non-standards conforming mode and sometimes the value 2 when extensions to the standard are enabled.

Coding Guidelines

The definition of this macro represents two pieces of information— (1) is definition as a macro, and (2) the tokens in its replacement list. Given that many vendors have chosen to always define the __STDC__ macro and use its replacement list to specify the details of the conformance status, it is the value of the replacement list rather than the status of being defined that provides reliable information.

Cg 2009.1

Tests of the conformance status of a translator, in source code, shall use the value of the replacement list of the __STDC__ macro, not its status as a defined macro.

2010 **__STDC_HOSTED__** The integer constant **1** if the implementation is a hosted implementation or the integer constant **0** if it is not.

__STDC_HOSTED macro

Commentary

The ability of an implementation to specify an accurate definition of the __STDC_HOSTED__ macro depends on tight integration between various phases of translation.

C90

Support for the __STDC_HOSTED__ macro is new in C99.

C++

Support for the __STDC_HOSTED__ macro is new in C99 and it is not available in C++.

Common Implementations

While some vendors only sell implementations targeted to purely a hosted or freestanding environment, a few vendors sell into both markets.

Coding Guidelines

Support for the __STDC_HOSTED__ macro is new in C99 and it is too early to tell whether there are any common translation phase dependency mistakes made by developers in its usage.

2011 **__STDC_VERSION__** The integer constant **199901L**.[153)]

__STDC_VERSION macro

Commentary

Because of existing, implementation, practice of giving values other than 1 to the __STDC__ macro, existing code (and implementations) would have been broken had this macro been used to denote the version of the standard supported. The simplest solution was to define a new macro that explicitly indicated the version of the standard supported by an implementation.

The integer constant 199901 has type **int** in some implementations. Specifying a suffix ensures the type of the integer constant denoted by this macro is always the same.

C90

Support for the __STDC_VERSION__ macro was first introduced in Amendment 1 to C90, where it was specified to have the value 199409L. In a C90 implementation (with no support for Amendment 1) occurrences of this macro are likely to be replaced by 0 (because it will not be defined as a macro).

1878 #if identifier replaced by 0

C++

Support for the __STDC_VERSION__ macro is not available in C++.

Other Languages

Perl supports the special variable $PERL_VERSION.

2012 **__TIME__** The time of translation of the preprocessing translation unit: a character string literal of the form **"hh:mm:ss"** as in the time generated by the **asctime** function.

__TIME__ macro

Commentary

There is no guarantee that the value for the __TIME__ macro will be obtained on the same day as the value for the __DATE__ macro. It is possible, for instance, for the value __TIME__ macro to be obtained first (returning "23:59:59") followed by the value of the __DATE__ macro (returning a date on the following day).

__DATE__ macro 2005

The discussion given for the __DATE__ macro is applicable here.

Common Implementations

The extent to which host platforms on which the translators execute maintains an accurately time of day can vary significantly. The realtime clock in many PCs often drifts by several seconds in a day. The internet time protocol, nntp, provides one method of ensuring that the error in the estimated current time does not become too large over long periods.

Coding Guidelines

A common use of the __TIME__ macro is to provide program build configuration management information (e.g., the time at which a source file was translated). As such great accuracy is rarely required. During development there may be many builds and build times becomes important if more than one is performed in a day. Issues such as the values of the macros __TIME__ and __DATE__ being synchronized to refer to the same day is a configuration management issue and is outside the scope of these coding guidelines.

If the time of translation is not available, an implementation-defined valid time shall be supplied. 2013

Commentary

__DATE__ date not available 2006

The discussion given for the __DATE__ macro also applies here.

The following macro names are conditionally defined by the implementation: 2014

Commentary

Specifying that these macro names are conditionally defined simplifies the writing of source that may need to be processed by C90 implementations (which are unlikely to have defined these macro names). The affect on program conformance status of using conditional features is discussed elsewhere.

footnote 96 2

C90

Support for conditionally defined macros is new in C99.

C++

Support for conditionally defined macros is new in C99 and none are defined in the C++ Standard.

Coding Guidelines

__STDC__ macro 2009

Like the __STDC__ macro these conditionally defined macros represent two pieces of information. However, these macros are not required to be defined by an implementation. The extent to which implementations will define these macros to have values other than those specified in the standard are not known. The very infrequent use of these macros (which may be because they are new, or because source that needs to make use of the information they provide are not yet common) a guideline recommendation similar to that given for the __STDC__ macro is not considered cost effective.

__STDC__ check replacement list 2009.1

__STDC_IEC_559__ macro

__STDC_IEC_559__ The integer constant **1**, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic). 2015

Commentary

The C Standard does not specify how conformance to the specification in annex F is to be measured.

C90

Support for the __STDC_IEC_559__ macro is new in C99.

C++

Support for the __STDC_IEC_559__ macro is new in C99 and it is not available in C++.

The C++ Standard defines, in the `std` namespace:

18.2.1.1

```
static const bool is_iec559 = false;
```

false is the default value. In the case where the value is **true** the requirements stated in C99 also occur in the C++ Standard. The member `is_iec559` is part of the numerics template and applies on a per type basis. However, the requirement for the same value representation, of floating types, implies that all floating types are likely to have the same value for this member.

2016 151) See "future language directions" (6.11.9). footnote 151

2017 152) The presumed source file name and line number can be changed by the **#line** directive. footnote 152

Commentary

This footnote clarifies the intent that implementations behave as if there is an association between evaluating a **#line** directive and the expanded form of subsequent occurrences of the __FILE__ __LINE__ macros.

C90

This observation is new in the C99 Standard.

C++

Like C90, the C++ Standard does not make this observation.

2018 153) This macro was not specified in ISO/IEC 9899:1990 and was specified as **199409L** in ISO/IEC 9899/AMD1:1995. footnote 153

Commentary

This is a brief history of the __STDC_VERSION__ macro. 2011 __STDC_VERS macro

---

2019 The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this International Standard.

Commentary

The type **long int** (or its unsigned version) is the only standard integer type supported by the C90 Standard and required to be capable of supporting the necessary range of values. 493 standard integer types

Coding Guidelines

In C99 the C committee made changes to the behavior, required by the C90 Standard, of strictly conforming programs. What behavior future revisions of the C Standard will specify is unknown, although the subsection on future language directions gives some warnings. 1000 operator () Future language directions

This macro is only really of practical use in checking that the translator being used supports a version of the C Standard that is greater than, or equal to, some known version.

2020 **__STDC_IEC_559_COMPLEX__** The integer constant **1**, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic). __STDC__IEC_559 macro

Commentary

The C Standard does not specify how adherence to the specification in annex G is to be measured (or how adherence differs from conformance). 2015 __STDC_IEC_ macro

C90

Support for the __STDC_IEC_559_COMPLEX__ macro is new in C99.

C++

Support for the `__STDC_IEC_559_COMPLEX__` macro is new in C99 and is not available in C++.

__STDC_ISO_10646__ macro

`__STDC_ISO_10646__` An integer constant of the form **`yyyymmL`** (for example, **`199712L`**). 2021

Commentary

ISO 10646 28 The wording was changed by the response to DR #273.

C90

Support for the `__STDC_ISO_10646__` macro is new in C99.

C++

Support for the `__STDC_ISO_10646__` macro is new in C99 and is not available in C++.

Other Languages

ISO 10646 28 The few languages (e.g., Java) that currently support ISO 10646 do not provide access to equivalent version information.

Common Implementations

The implementation shipped with RedHat Linux version 9 has the value 200009L.

If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character. 2022

Commentary

This is a requirement on an implementation that defines the `__STDC_ISO_10646__` macro. A short identifier short identifier 818 is representable in eight hexadecimal digits (i.e., values in the range `0..4294967295`). For instance:

```
1  L'\u00A3' == L'\x00A3'            if __STDC_ISO_10646__ defined
2  L'\u00A3' == L'£'                 always true
3  L'\x00A3' == (wchar_t) 0x00A3     always true
4  L'\x00A3' == L'£'                 if __STDC_ISO_10646__ defined
5  L'£' == (wchar_t) 0x00A3          if __STDC_ISO_10646__ defined
```

The term *Unicode required set* is defined in the following C sentence.

This wording was changed by the response to DR #273.

C90

This form of encoding was not mentioned in the C90 Standard.

C++

This form of encoding is not mentioned in the C++ Standard.

Unicode required set

The *Unicoderequiredset* consists of all the characters that are ~~intended to indicate that values of type **`wchar_t`** are the coded representations of the characters~~ defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. 2023

Commentary

This gives permission for implementations to support amendments and technical corrigenda to ISO/IEC ISO 10646 28 10646 that are published after the C99 Standard. However, all documents published on or before the specified month must be supported.

The wording was changed by the response to DR #273.

Coding Guidelines

Program dependencies on the version of ISO/IEC 10646 is a configuration management issue that is outside the scope of these guidelines.

2024 **__STDC_MB_MIGHT_NEQ_WC__** The integer constant **1**, intended to indicate that, in the encoding for **wchar_t**, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.

Commentary

Until the publication of TC2, implementations were required to support the equality `'x' == L'x'`, where x is any member of the basic character set. This requirement restricted an implementation's choice of encoding for the type `wchar_t`. The response to DR #279 removed this restriction, but did not provide a mechanism for developers to write code that checked the behavior of their implementation (a large body of existing code relies on pre-TC2 guaranteed behavior).

L'x' == 'x'
215 basic character set
62 wide character

The expression `L'\xhh' == '\xhh'` is always true (as long as 0xhh is less than UCHAR_MAX). Developers would be very confused if this equality was true for escape sequences, but was not true when the escape sequences were replaced by members of the basic execution character set having the same numeric value (in a particular character set, for instance Ascii).

This sentence was added by the response to DR #333 and provides a specification for the pre-defined macro name introduced by the response to DR #321 (and supported by the Austin Group).

2025 The values of the predefined macros (except for **__FILE__** and **__LINE__**) remain constant throughout the translation unit.

predefined macros constant values during translation

Commentary

This is a requirement on the implementation. The standard places no requirements on exactly when the implementation assigns a value to the `__DATE__` and `__TIME__` macros.

The predefined macros `__DATE__` and `__TIME__` are intended to provide configuration management information about when the source was translated. Having to deal with time differences caused by the finite time needed to translate a source file would be an unnecessary complication for developers.

Given a sufficiently high-performance processor and fast translation, or a translation environment where no date and time information is available, it is possible that the values of the `__DATE__` and `__TIME__` macros will be the same during translations of different source files.

Common Implementations

Many implementations allow more than one source file to be specified, for translation, at the same time. The standard is silent about the values of these predefined macros, across multiple sources files, during a single invocation of a translator.

10 program transformation mechanism
120 footnote 5

Many implementations use several programs to translate a source file, preprocessing being handled by one of these programs. The details of invoking these separate programs is handled by a controlling program that provides the user interface, including breaking the translation of multiple source files into a sequence of translations of individual source files. Given this implementation strategy it is very likely that the preprocessor will be invoked separately for every source file being processed (with the value of the predefined macros being set during the startup of this preprocessor program) and can be different between different invocations.

2026 None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive.

predefined macros not #defined

Commentary

This wording covers some of the possible source code constructs where **defined** can occur, but not all. For instance:

```
1  #define f(defined) defined
2
3  x = f(1);
4
```

```
5  #if f(1)
6  /* ... */
7  #endif
8
9  #if defined(defined)
10 /* ... */
11 #endif
```

Rationale If the identifier **defined** were to be defined as a macro, `defined(X)` would mean the macro expansion in C text proper and the operator expression in a preprocessing directive (or else that the operator would no longer be available). To avoid this problem, such a definition is not permitted (§6.10.8).

C++

The C++ Standard uses different wording that has the same meaning.

16.8p3 *If any of the pre-defined macro names in this subclause, or the identifier* ***defined****, is the subject of a* ***#define*** *or a* ***#undef*** *preprocessing directive, the behavior is undefined.*

Common Implementations

Some, but not all, implementations allow predefined macros to be **#undef**ed and for the identifier **defined** to be defined as a macro name.

Coding Guidelines

The usage described is very rare and for this reason a guideline recommendation is not considered cost effective.

---

macro name predefined reserved

Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore. 2027

Commentary

This is a requirement on the implementation. It also acts as a warning to developers that macro names beginning with these sequences of characters may be reserved by an implementation. A predefined macro is generally considered to be one that is defined by a translator prior to the start of translation, i.e., it is not necessary for the source file to **#include** a particular header for it to be defined.

C++

The C++ Standard does not reserve names for any other predefined macros.

Common Implementations

The definition of some predefined macros may be affected by command line options passed to the translator. For instance, specifying the version of a particular processor architecture to generate machine code for. Such version specific architecture macro definitions might then affect which conditional inclusion directives are processed, within implementation supplied headers.

Although the standard reserves identifiers having these spellings for use by implementations, most implementations also use identifiers having spellings outside of this set, for their own internal use.

__cplusplus

The implementation shall not predefine the macro **__cplusplus**, nor shall it define it in any standard header. 2028

Commentary

This is a requirement on the implementation. It has become established practice to use this macro name to distinguish those parts of a header that are specific to C++ only and should be ignored by a C translator.

C90

This requirement was not specified in the C90 Standard. Given the prevalence of C++ translators, vendors were aware of the issues involved in predefining such a macro name (i.e., they did not do it).

C++

The name __cplusplus is defined to the value 199711L when compiling a C++ translation unit.[143)] 16.8p1

Other Languages

This macro name is unique to C in the sense that this language is sufficiently similar to the C++ language for implementors to be able to share headers in implementations of both languages.

2029 **Forward references:** the **asctime** function (7.23.3.1), standard headers (7.1.2).

6.10.9 Pragma operator

Semantics

2030 A unary operator expression of the form: _Pragma operator

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting the **L** prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash.

Commentary

This defines the term *destringized*. This operation the inverse of the operation performed by the stringize operator. 1954 # escape sequence handling

As an alternative syntax for a **#pragma** directive, the **_Pragma** operator has the advantage that it can be used in a macro replacement list. If a translator is directed to produce a preprocessed version of the source file, then expressions involving the unary **_Pragma** operator and **#pragma** directives should be treated consistently in whether they are preserved and in whether macro invocations within them are expanded. Rationale

The prior art on which this directive was based comes from the Cray Standard C compiler (see WG14/N449).

C90

Support for the **_Pragma** unary operator is new in C99.

C++

Support for the **_Pragma** unary operator is new in C99 and it is not available in C++.

Coding Guidelines

Support for the **_Pragma** operator is new in C99 and at the time of this writing there is insufficient experience available in its use to know whether any guideline recommendation is worthwhile.

Example

The **_Pragma** operator can be used to reduce the visual clutter in source code. For instance, rather than duplicating a sequence of conditional inclusion directives, controlling a **pragma** directive, in the various source files or function definitions that require them, they can be encapsulated in a single macro definition.

```
#if defined(Machine_A)
   #define IZATION_HINT _Pragma("ivdep")       /* Ignore vector dependencies. */
#elif defined(Machine_B)
   #define IZATION_HINT _Pragma("independent") /* Iterations are independent. */
#endif

void N449_a(int n, double * a, double * b)
{
#if defined(Machine_A)
   #pragma ivdep            /* Vectorization hint (ignore vector dependencies).   */
#elif defined(Machine_B)
   #pragma independent      /* Parallelization hint (iterations are independent). */
#endif
while (n-- > 0)
   {
   *a++ += *b++;
   }
}

void N449_b(int n, double * a, double * b)
{
IZATION_HINT
while (n-- > 0)
   {
   *a++ += *b++;
   }
}
```

The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. 2031

Commentary

The string literal, or any macro invocations used to create it, will have already been processed by translation phases 1–3. Consequently the resulting sequence of characters will not include any new-line characters (because a string literal cannot them) or trigraph sequences (they will have been replaced in translation phase 1 and the escape sequence processing that might create new trigraph sequences does not occur until translation phase 5).

string literal 895 syntax
trigraph sequences 117 phase 1
translation phase 133 5

The original four preprocessing tokens in the unary operator expression are removed. 2032

Commentary

Exactly when this removal occurs, within translation phase 4 (other preprocessing tokens are deleted at the end of translation phase 4), is not specified. Because defining **_Pragma** as a macro name (e.g., `#define _Pragma func_call`) results in undefined behavior it is not possible for the timing of the removal to affect the output of a strictly conforming program.

preprocessing directives 132 deleted

EXAMPLE A directive of the form: 2033

EXAMPLE _Pragma

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING ( ..\listing.dir )
```

Commentary

The macro expansion sequence is:

```
1  LISTING ( ..\listing.dir )
2  PRAGMA(listing on "..\listing.h")
3  _Pragma("listing on \"..\\listing.h\"")
```

with the character `.` being treated as a "each non-white-space character that cannot be one of the above". 770 preprocessing token syntax

6.11 Future language directions

Commentary

Future language directions

This subclause includes specific mention of the future direction in which the Committee intends to extend and/or restrict the language. The contents of this subclause should be considered as quite likely to become a part of the next version of the Standard. Implementors are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming implementation. Users are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming program. Rationale

6.11.1 Floating types

2034 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **`long double`**.

floating types future language directions

Commentary

Unlike the integer types, the standard does not specify any mechanism for implementations to provide additional floating-point types. 482 extended signed integer types

C90

This future direction is new in C99.

C++

The C++ Standard specifies (Annex D) deprecated features. With one exception these all relate to constructs specific to C++.

Each C header, whose name has the form `name.h`*, behaves as if each name placed in the Standard library namespace by the corresponding* `cname` *header is also placed within the namespace scope of the namespace* `std` *and is followed by an explicit* `using-declaration` *(7.3.3)* D.5p2

Common Implementations

While some processors support floating types having 128 bits value bits,[571] support for 256 bits (known as *quad-double*) is currently only available in software.[572]

Coding Guidelines

Some existing programs were broken by the introduction, in C99, of an integer type that was *bigger* than **`long`**. This situation occurred because of an assumption made by developers (and at times the C committee). The introduction of a floating-point type with greater range and precision than **`long double`** may cause existing programs to break for the same reason. However, it is not possible to estimate the costs and benefits of taking account of this possibility, when writing or modifying code, in guideline recommendations aimed at a wide audience. Developers are left to make their own cost/benefit analysis.

6.11.2 Linkages of identifiers

identifier linkage
future language directions

Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature. 2035

Commentary

static 425
internal linkage

Declaring an identifier at file scope using the **static** storage-class specifier gives it internal linkage. Although, in C99, subsequent declarations of the same identifier may omit the storage-class specifier (it either has internal linkage, or the behavior is undefined), future revisions of the standard may not support this usage.

function 430
no storage-class
object 431
file scope no storage-class

C90

This future direction is new in C99.

Usage

The translated form of this book's benchmark programs contained 12 declarations of an identifier with internal linkage at file scope without the **static** storage-class specifier.

6.11.3 External names

significant characters
future language directions

Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations. 2036

Commentary

internal 282
identifier
significant characters

The issues surrounding this restriction are discussed elsewhere.

C90

Part of the future language direction specified in C90 was implemented in C99.

Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

escape sequences
future language directions

Lowercase letters as escape sequences are reserved for future standardization. 2037

Commentary

escape sequence 866
syntax

This position has not changed since publication of C90. The Committee received requests to support additional escape sequences in C99, but with one exception (i.e., \u) turned them down.

Other characters may be used in extensions. 2038

Commentary

The Committee chose to support \U in C99.

6.11.5 Storage-class specifiers

storage-class specifiers
future language directions

The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. 2039

Commentary

This position has not changed since publication of C90.

Rationale

The practice of placing the storage class specifier other than first in a declaration was branded obsolescent. The Committee felt it desirable to rule out such constructs as

```
enum { aaa, aab,
    /* etc. */
zzy, zzz } typedef a2z;
```

in some future standard.

This issue is also discussed elsewhere.

1357 declaration specifiers

Other Languages

This restriction is required by the syntax of many programming languages.

6.11.6 Function declarators

2040 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

function declarators future language directions

Commentary

This position has not changed since the publication of the C90 Standard.

Rationale

It was obviously out of the question to remove syntax used in the overwhelming majority of extant C code, so the Standard specifies two ways of writing function declarations and function definitions. Characterizing the old style as obsolescent is meant to discourage its use and to serve as a strong endorsement by the Committee of the new style. It confidently expects that approval and adoption of the prototype style will make it feasible for some future C Standard to remove the old style syntax.

Removing support for this form of declaration could also impact its definition.

2041 function definitions future language directions

Coding Guidelines

If the guideline recommendation specifying the use of function prototypes is followed then any future change will not be an issue.

1810.1 function declaration use prototype

6.11.7 Function definitions

2041 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

function definitions future language directions

Commentary

This position has not changed since the publication of the C90 Standard. The issues are the same as for function declarators.

2040 function declarators future language directions

6.11.8 Pragma directives

2042 Pragmas whose first preprocessing token is `STDC` are reserved for future standardization.

Pragma directives future language directions

Commentary

The Committee is claiming a very limited subset of possible **`pragma`** directives for their own future use. However, this does not mean that any future standard **`pragma`** directives will have `STDC` as the third preprocessing token.

C90

Support for this form of **`pragma`** directive is new in C99.

6.11.9 Predefined macro names

Predefined macro names future language directions

Macro names beginning with `__STDC_` are reserved for future standardization. 2043

Commentary

The C90 Standard reserved identifiers that began with two underscores in the library. This specification appears in a language clause and would imply that this set of spellings is reserved for use by the preprocessor (perhaps for predefined macro names).

C90

The specification of this reserved set of macro name spellings is new in C99.

References

1. T. Aamodt and P. Chow. Numerical error minimizing floating-point to fixed-point ANSI C compilation. In 1^{st} *Workshop on Media Processors and Digital Signal Processing*, Nov. 1999.
2. P. L. Ackerman and E. D. Heggestad. Intelligence, personality, and interests: Evidence for overlapping traits. *Psychological Bulletin*, 121(2):219–245, 1997.
3. E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.
4. P. A. Adams and J. K. Adams. Confidence in the recognition and reproduction of words difficult to spell. *American Journal of Psychology*, 73:544–552, 1960.
5. V. Agarwal, M. S. Hrishikesh, S. W. K. Doug, and Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture*, 2000.
6. Agere Systems. *DSP16000 Digital Signal Processor Core Information Manual*. Agere Systems, mn02-026winf edition, June 2002.
7. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.
8. A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, 1976.
9. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1985.
10. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *SIGPLAN Conference on Programming Language Design and Implementation PLDI'03*, pages 129–140, June 2003.
11. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 266–277, Los Altos, CA 94022, USA, Sept. 1999. Morgan Kaufmann Publishers.
12. M. M. Al-Jarrah and I. S. Torsun. An empirical analysis of COBOL programs. *Software–Practice and Experience*, 9:341–359, 1979.
13. M. J. Alexander, M. W. Bailey, B. R. Childers, J. W. Davidson, and S. Jinturkar. Memory bandwidth optimizations for wide-bus machines. Technical Report CS-92-24, Department of Computer Science, University of Virginia, Aug. 4 1992.
14. F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.
15. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
16. P. A. Allen, B. Wallace, and T. A. Weber. Influence of case type, word frequency, and exposure duration on visual word recognition. *Journal of Experimental Psychology: Human Perception and Performance*, 21(4):914–934, 1995.
17. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture*. Morgan Kaufmann Publishers, 2002.
18. E. M. Altmann. Near-term memory in programming: A simulation-based analysis. *International Journal of Human-Computer Studies*, 54:189–210, 2001.
19. E. M. Altmann. Functional decay of memory for tasks. *Psychological Research*, 66(4):287–297, 2002.
20. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
21. C. Álvarez, J. Corbal, E. Salamí, and M. Valero. On the potential of tolerant region reuse for multimedia applications. In *Proceedings of the* 15^{th} *international conference on Supercomputing*, pages 218–228. ACM Press, Apr. 2001.
22. C. J. Álvarez, M. Carreiras, and M. de Vega. Syllable-frequency effect in visual word recognition: Evidence of sequential-type processing. *Psicológica*, 21:341–374, 2000.
23. R. Alverson. Integer division using reciprocals. In P. Kornerup and D. W. Matula, editors, *Proceedings of the* 10^{th} *IEEE Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.
24. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. *1990 International Conference on Supercomputing*, June 11-15 1990. Published as Computer Architecture News 18:3.
25. L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, Oct. 2000.
26. AMD. *Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*. Advanced Micro Devices, Inc, 3.03 edition, Sept. 2002.
27. AMD. *AMD64 Architecture Programmer's Manual Volume 1: Application programming*. Advanced Micro Devices, Inc, 3.09 edition, Sept. 2003.
28. Z. Ammaraguellat. A control-flow normalization algorithm and its complexity. *Software Engineering*, 18(3):237–251, Mar. 1992.
29. Analog Devices, Inc. *ADSP-21065L SHARC DSP Technical Reference*. Analog Devices, Inc, 2.0 edition, July 2003.
30. Analog Devices, Inc. *C/C++ Compiler and Library Manual for SHARC*. Analog Devices, Inc, 4.0 edition, Jan. 2003.
31. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
32. J. R. Anderson. Interference: The relationship between response latency and response accuracy. *Journal of Experimental Psychology: Human Learning and Memory*, 7(5):326–343, 1981.
33. J. R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, fifth edition, 2000.
34. J. R. Anderson. *Learning and Memory*. John Wiley & Sons, Inc, second edition, 2000.
35. J. R. Anderson and C. Libiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.
36. J. R. Anderson and R. Milson. Human memory: An adaptive perspective. *Psychological Review*, 96(4):703–719, 1989.
37. S. Andrews. Lexical retrieval and selection processes: Effects of transposed-letter confusability. *Journal of Memory and Language*, 35:775–800, 1996.
38. S. Andrews. The effect of orthographic similarity on lexical retrieval: Resolving neighborhood conflicts. *Psychonomic Bulletin & Review*, 4(4):439–461, 1997.
39. S. Andrews and D. R. Scarratt. Rule and analogy mechanisms in reading nonwords: Hough dou peapel rede gnew wirds? *Journal of Experimental Psychology: Human Perception and Performance*, 24(4):1052–1086, 1998.

40. Ángel Fernández and M. A. Alonso. The relative value of environmental context reinstatement in free recall. *Psicológica*, 22:253–266, 2001.
41. Anon. How genes work: A quick guide to life's operating system. *The Economist*, June 29 2000.
42. Anon. OpenMP C and C++ application program interface. Technical Report Version 2.0, OpenMP Architecture Review Board, Mar. 2002.
43. Anon. The international obfuscated C code contest. www.ioccc.org, 2003.
44. Anon. Tendra home page. www.tendra.org, 2003.
45. Anon. Top 500 supercomputer sites. www.top500.org, 2003.
46. Anon. Trimaran home page. www.trimaran.org, 2003.
47. N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON'98*, pages 213–222, 1998.
48. N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 84–93. IEEE Computer Society Press/ACM Press, 1998.
49. N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.
50. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
51. A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. Technical Report TR-630-00, Princeton University, Mar. 2001.
52. Apple Computer. *Inside Macintosh: PowerPC Numerics*. Addison–Wesley, 1994.
53. K. D. Arbuthnott and J. I. D. Campbell. Effects of operand order and problem repetition on error priming in cognitive arithmetic. *Canadian Journal of Experimental Psychology*, 50(2):182–195, 1996.
54. H. R. Arkes, R. M. Dawes, and C. Christensen. Factors influencing the use of a decision rule in a probabilistic task. *Organizational Behavior and Human Decision Processes*, 37:93–110, 1986.
55. ARM. *ARM Developer Suite: Compilers and Libraries Guide*. ARM Limited, 1.2 edition, Nov. 2001.
56. ARM. *ARM1026EJ-S: Technical Reference Manual*. ARM Limited, r0p1 edition, Dec. 2002.
57. B. Armstrong and R. Eigenmann. Performance forecasting: Characterization of applications on current and future architectures. Technical Report ECE-HPCLab-97202, Purdue University School of ECE, Jan. 1997.
58. M. Arnold and H. Corporaal. Data transport reduction in move processors. In *Third Annual Conference of the Advance School for Computing and Imaging*, pages 68–75, June 1997.
59. M. Arnold, S. J. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Addative Compilation and Optimization (DYNAMO'00)*, pages 52–64. ACM, Jan. 2000.
60. M. H. Ashcraft. Cognitive arithmetic: A review of data and theory. *Cognition*, 44:75–106, 1992.
61. E. A. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs. Technical Report CS-TR-71-188, Stanford University, Department of Computer Science, Jan. 1971.
62. R. L. Ashenhurst and N. Metropolis. Unnormalized floating point arithmetic. *Journal of the ACM*, 6(3):415–428, July 1959.
63. D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 324–333, New York, May 1999. Association for Computing Machinery.
64. AT&T Document Management. *WE DSP32 Digital Signal Processor Information Manual*, 1988.
65. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.
66. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.
67. A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.
68. R. H. Baayen. *Word Frequency Distributions*. Kluwer Academic Publishers, 2001.
69. J. Backus. The history of FORTRAN I, II, and III. *SIGPLAN Notices*, 13(8):165–180, 1978.
70. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, University of California, Berkeley, 1993.
71. A. Baddeley. Working memory: Looking back and looking forward. *Nature Reviews*, 4(10):829–839, Oct. 2003.
72. A. Baddeley, M. Conway, and J. Aggleton. *Episodic Memory: New Directions in Research*. Oxford University Press, 2002.
73. A. D. Baddeley. How does acoustic similarity influence short-term memory? *Quarterly Journal of Experimental Psychology*, 20:249–264, 1968.
74. A. D. Baddeley. Language habits, acoustic confusability, and immediate memory for redundant letter sequences. *Psychonomic Science*, 22(2):120–121, 1971.
75. A. D. Baddeley. *Essentials of Human Memory*. Psychology Press, 1999.
76. A. D. Baddeley, N. Thomson, and M. Buchanan. Word length and the structure of short-term memory. *Journal of Verbal Learning and Verbal Behavior*, 14:575–589, 1975.
77. R. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison–Wesley, Reading, MA, USA, 1990.
78. P. Baggett and A. Ehrenfeucht. How an unfamiliar thing should be called. *Journal of Psycholinguistic Research*, 11(5):437–445, 1982.
79. I. Bahar, B. Calder, and D. Grunwald. A comparison of software code reordering and victim buffers. *ACM SIGARCH Computer Architecture News*, Mar. 1999.
80. H. P. Bahrick. Semantic memory content in permastore: Fifty years of memory for Spanish learned in school. *Journal of Experimental Psychology: General*, 113(1):1–26, 1984.
81. J. N. Bailenson, M. S. Shum, and J. D. Coley. A bird's eye view: Biological categorization and reasoning within and across cultures. *Cognition*, 84:1–53, 2002.

82. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.
83. J. L. Bailey and G. Stefaniak. Industry perceptions of the knowledge, skills, and abilities needed by computer programmers. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research (SIGCPR 2001)*, pages 93–99. ACM Press, 2001.
84. T. M. Bailey and U. Hahn. Phoneme similarity and confusability. *Journal of Memory and Language*, 52(???):339–362, 2005.
85. B. S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, July 1995. IEEE Computer Society Press.
86. M. C. Baker. *The Atoms of Language*. Basic Books, 2001.
87. C. Y. Baldwin and K. B. Clark. *Design Rules. Volume 1. The Power of Modularity*. The MIT Press, 2000.
88. T. Ball and J. R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, June 1993.
89. U. Banerjee. *Dependence analysis for supercomputing*. The Kluwer international series in engineering and computer science. Parallel processing and fifth generation computing. Kluwer Academic, Boston, MA, USA, 1988.
90. R. D. Banker and S. A. Slaughter. The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, 11(3):219–240, Sept. 2000.
91. S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International conference on Architectural support for programming languages and operating systems*, pages 394–403, Apr. 2006.
92. P. Banyard and N. Hunt. Something missing? *The Psychologist*, 13(2):68–71, 2000.
93. J. R. Barclay, J. D. Bransford, J. J. Franks, N. S. McCarrell, and K. Nitsch. Comprehension and semantic flexibility. *Journal of Verbal Learning and Verbal Behavior*, 13:471–481, 1974.
94. J. H. Barkow, L. Cosmides, and J. Tooby. *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*. Oxford University Press, 1992.
95. C. Barry and P. H. K. Seymour. Lexical priming and sound-to-spelling contingency effects in nonword spelling. *The Quarterly Journal of Experimental Psychology*, 40A(1):5–40, 1988.
96. R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures*. PhD thesis, M.I.T., Jan. 2000.
97. V. R. Basili, L. C. Briand, and S. Morasca. Defining and validating measures for object-based high-level design. Technical Report IESE-Report No. 018.98/E, Fraunhofer Institute for Experimental Software Engineering, 1998.
98. V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering: An International Journal*, 1(2):133–164, 1996.
99. A. Bauer. Compilation of functional programming languages using GCC - Tail calls. Thesis (m.s.), Munich University of Technology, Jan. 2003.
100. L. Bauer. *English Word-formation*. Cambridge University Press, 1983.
101. L. Bauer. *An Introduction to International Varieties of English*. Edinburgh University Press, 2002.
102. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–378. IEEE Computer Society Press, 1998.
103. K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES'01*, pages 203–210. ACM Press, Nov. 2001.
104. BCS SIGIST. Standard for software component testing. Technical Report Working Draft 3.4, British Computer Society and Special Interest Group in Software Testing, Apr. 2001.
105. L. R. Beach, V. E. Barnes, and J. J. J. Christensen-Szalanski. Beyond heuristics and biases: A contingency model of judgmental forecasting. *Journal of Forecasting*, 5:143–157, 1986.
106. C. Beauvillain, K. Doré, and V. Baudouin. The 'center of gravity' of words: Evidence for an effect of the word-initial letters. *Vision Research*, 36(4):589–603, 1996.
107. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990.
108. J. P. Beldie, S. Pastoor, and E. Schwarz. Fixed versus variable letter width for television text. *Human Factors*, 25(3):273–277, 1983.
109. G. Bell and J. Gray. High performance computing: Crays, clusters, and centers. what next? Technical Report MSR-TR-2001-76, Microsoft Research, Sept. 2001.
110. G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of silent stores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 133–144. IEEE, Oct. 2000.
111. V. A. Bell and P. N. Johnson-Laird. A model theory of modal reasoning. *Cognitive Science*, 22(1):25–51, 1998.
112. M. E. Benitez and J. W. Davidson. Register deprivation measurements. Technical Report CS-93-63, Department of Computer Science, University of Virginia, Nov. 15 1993.
113. M. E. Benitez and J. W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report Technical Report CS-94-42, University of Virginia, 1994.
114. Y. Benkler. Coase's penguin, or, Linux and the nature of the firm. *The Yale Law Journal*, 112(3), Dec. 2002.
115. B. Berlin and P. Kay. *Basic Color Terms*. Berkeley: University of California Press, 1969.
116. R. S. Berndt, J. A. Reggia, and C. C. Mitchum. Empirical derived probabilities for grapheme-to-phoneme correspondences in English. *Behavior and Research Methods, Instruments, & Computers*, 19(1):1–9, 1987.
117. D. C. Berry and D. E. Broadbent. On the relationship between task performance and associated verbalized knowledge. *Quarterly Journal of Experimental Psychology*, 36A:209–231, 1984.
118. R.-J. Beun and A. H. M. Cremers. Multimodal reference to objects: An empirical approach. In H. Bunt and R.-J. Beun, editors, *Cooperative Multimodal Communication*, pages 64–88. Springer-Verlag, 1998.
119. R. Bhargava, J. Rubio, S. Kannan, and L. K. John. Understanding the impact of X86/NT computing on microarchitecture. In L. K. John and A. M. G. Maynard, editors, *Characterization of Contemporary Workloads*, chapter 10, pages 203–228. Kluwer Academic Publishers, 2001.

120. S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. Technical Report CS-TR-4063, University of Maryland, College Park, Sept. 1999.

121. D. Biber, S. Johansson, G. Leech, S. Conrad, and E. Finegan. *Longman Grammar of Spoken and Written English*. Pearson Education, 1999.

122. J. M. Bieman and V. Murdock. Finding code on the world wide web: A preliminary investigation. In *Proceedings First International Workshop on Source Code Analysis and Manipulation (SCAM2001)*, pages 73–78, 2001.

123. A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, May 1996.

124. S. Bikhchandani, D. Hirshleifer, and I. Welch. A theory of fads, fashion, custom, and cultural change as information cascades. *Journal of Political Economy*, 100(5):992–1026, 1992.

125. S. Blackmore. *The Meme Machine*. Oxford University Press, 1999.

126. M. S. Blaubergs and M. D. S. Braine. Short-term memory limitations on decoding self-embedded sentences. *Journal of Experimental Psychology*, 102(4):745–748, 1974.

127. S. Blazey, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: International Symposium on Formal Methods*, pages 460–475. Springer-Verlag, 2006.

128. J. Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, 171:156–182, 2001.

129. J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 11(2):115–144, 1996.

130. B. E. Bliss. Instrumentation of FORTRAN programs for automatic roundoff error analysis and performance evaluation. Thesis (m.s.), University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1990.

131. R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.

132. P. P. G. Boersma. *Functional Phonology*. PhD thesis, University of Amsterdam, Sept. 1998.

133. C. Boiarsky. Consistency of spelling and pronunciation deviations of Appalachian students. *Modern Language Journal*, 53:347–350, 1969.

134. D. B. Boles and J. E. Clifford. An upper- and lowercase alphabetic similarity matrix, with derived generation similarity values. *Behavior Research Methods, Instruments, & Computers*, 21(6):579–586, 1989.

135. D. Bolinger and T. Bronson. *Applying RCS and SCCS*. O'Reilly & Associates, Inc, 1995.

136. T. Bonk and U. Rüde. Performance analysis and optimization of numerically intensive programs. Technical Report SFB Bericht 342/26/92 A, Technische Universität München, Germany, Nov. 1992.

137. G. Boole. *An Investigation of the Laws of Thought*. Dover Publications, 1973.

138. D. J. Boorstin. *The Discoverers*. Phoenix Press, 1983.

139. L. Boroditsky. Metaphoric structuring: understanding time through spatial metaphors. *Cognition*, 75:1–28, 2000.

140. L. Boroditsky. Does language shape thought?: Mandarin and English speaker' conception of time. *Cognitive Psychology*, 43:1–22, 2001.

141. P. Boucher. Teaching compound nouns in English: an application of the Charlie Brown principle. *CIEREC/TRAVAUX LXXVI*, pages 33–50, 1992.

142. P. Boucher, F. Danna, and P. Sébillot. Compounds: An intelligent tutoring system for learning to use compounds in English. Technical Report Publication Interne No. 718, IRISA, Campus Universitaire de Beaulieu, France, Apr. 1993.

143. H. Bouma. Visual recognition of isolated lower-case letters. *Vision Research*, 11:459–474, 1971.

144. C. P. Bourne. Frequency and impact of spelling errors in bibliographic data bases. *Information Processing & Management*, 13(1):1–12, 1997.

145. C. P. Bourne and D. F. Ford. A study of methods for systematically abbreviating English words and names. *Journal of the ACM*, 8(4):538–552, 1961.

146. G. H. Bower, J. B. Black, and T. J. Turner. Scripts in memory for text. *Cognitive Psychology*, 11:177–220, 1979.

147. G. H. Bower, M. C. Clark, A. M. Lesgold, and D. Winzenz. Hierarchical retrieval schemes in recall of categorized word lists. *Journal of Verbal Learning and Verbal Behavior*, 8:323–343, 1969.

148. R. L. Bowman, E. J. Ratliff, and D. B. Whalley. Decreasing process memory requirements by overlapping program portions. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 115–124, Jan. 1998.

149. M. G. Bradac, D. E. Perry, and L. G. Votta. Prototyping A process monitoring experiment. *IEEE Transactions on Software Engineering*, 20(10):774–784, 1994.

150. G. L. Bradshaw and J. R. Anderson. Elaborative encoding as an explanation of levels of processing. *Journal of Verbal Learning and Verbal Behavior*, 21:165–174, 1982.

151. M. D. S. Braine and D. P. O'Brian. A theory of *if*: A lexical entry, reasoning program, and pragmatic principles. *Psychological Review*, 98(2):182–203, 1991.

152. J. D. Bransford and J. J. Franks. The abstraction of linguistic ideas. *Cognitive Psychology*, 2:331–350, 1971.

153. J. D. Bransford and M. K. Johnson. Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behavior*, 11:717–726, 1972.

154. R. A. Brealey and S. C. Myers. *Principles of Corporate Finance*. Irwin McGraw–Hill, 2000.

155. S. Brédart. Retrieval failure in face naming. *Memory*, 1:351–366, 1993.

156. E. J. Breen. *Extensible Interactive C*. eic.sourceforge.net, June 2000.

157. T. Brennen. The difficulty with recalling people's names: The plausible phonology hypothesis. *Memory*, 1(4):409–431, 1993.

158. R. P. Brent. On the precision attainable with various floating-point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.

159. P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Apr. 1992.

160. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software–Practice and Experience*, 26(6):701–724, 1997.

161. S. Broadbent. Font requirements for next generation air traffic management systems. Technical Report HRS/HSP-006-REP-01, European Organisation for the Safety of Air Navigation, 2000.

162. A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Technical Report ISERN-96-10, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1996.

163. D. Brooks and M. Martinosi. Value-based clock gating and operation packing: Dynamic strategies for improving processor power and performance. *ACM Transactions on Computer Systems*, 18(2):89–126, May 2000.

164. L. Brooks. Visual pattern in fluent word identification. In A. S. Reber and D. L. Scarborough, editors, *Toward a Psychology of Reading: The Proceedings of the CUNY Conference*, chapter 3, pages 143–181. Erlbaum, 1977.

165. J. O. Brooks III, L. Friedman, J. M. Gibson, and J. A. Yesavage. Spontaneous mnemonic strategies used by older and younger adults to remember proper names. *Memory*, 1(4):393–407, 1993.

166. F. P. Brooks, Jr. *The Mythical Man-Month*. Addison–Wesley, anniversary edition, 1995.

167. H. D. Brown. Categories of spelling difficulty in speakers of English as a first and second language. *Journal of Verbal Learning and Verbal Behavior*, 9:232–236, 1970.

168. J. Brown, V. J. Lewis, and A. F. Monk. Memorability, word frequency and negative recognition. *Quarterly Journal of Experimental Psychology*, 29:461–473, 1977.

169. P. J. Brown. *The ML/I macro processor*. University of Kent at Canterbury, fourth edition, Aug. 1970.

170. P. J. Brown. *Macro Processors and Techniques for Portable Software*. John Wiley & Sons, 1974.

171. R. W. Brown, A. H. Black, and A. E. Horowitz. Phonetic symbolism in natural languages. *Journal of Abnormal and Social Psychology*, 50:388–393, 1955.

172. J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, 1976.

173. J. L. Bruno and T. Lassagne. The generation of optimal code for stack machines. *Journal of the ACM*, 22(3):382–396, 1975.

174. M. Brysbaert. Arabic number reading: On the nature of the numerical scale and the origin of phonological recoding. *Journal of Experimental Psychology: General*, 124(4):434–452, 1995.

175. L. Buchanan, N. R. Brown, R. Cabeza, and C. Maitson. False memories and semantic lexicon arrangement. *Brian and Language*, 68:172–177, 1999.

176. W. Buchholz. Origin of the term *byte*. *Annals of the History of Computing*, 3(1):72–72, 1981.

177. A. Budanitsky. Lexical semantic relatedness and its application in natural language processing. Technical Report CSRG-390, Computer Systems Research Group, University of Toronto, Aug. 1999.

178. T. Budd. *An APL Compiler*. Springer-Verlag, 1988.

179. H. Bull. *Multics C User's Guide*. Honeywell Bull, Inc, 1987.

180. H. Bunke. A fast algorithm for finding the nearest neighbor of a word in a dictionary. Technical Report IAM-93-025, Institut für Informatik, Nov. 1993.

181. D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In 23^{rd} *Annual International Symposium on Computer Architecture*, pages 78–89, 1996.

182. C. Burgess and K. Livesay. The effect of corpus size on predicting reaction time in a basic word recognition task: Moving on from Kučera and Francis. *Behavior Research Methods, Instruments, & Computers*, 30(2):272–277, 1998.

183. D. M. Burke, L. Peters, and R. M. Harrold. Word association norms for young and older adults. *Social and Behavioral Science Documents*, 17(2), 1987.

184. J. Burlin. Optimizing stack frame layout for embedded systems. Thesis (m.s.), Uppsala University, Information Technology Computer Science Department, Nov. 2000.

185. Q. L. Burrell. A note on ageing in a library circulation model. *Journal of Documentation*, 41(2):100–115, 1985.

186. M. Burtscher. *Improving Context-Based Load Value Prediction*. PhD thesis, University of Colorado, 2000.

187. M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, June 2002.

188. M. D. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.

189. B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.

190. B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. G. Zorn. Corpus-based static branch prediction. *SIGPLAN Notices*, 30(6):79–92, June 1995.

191. B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. Technical Report Research Report 95/6, Western Research Laboratory - Compaq, 1995.

192. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.

193. C. F. Camerer and E. F. Johnson. The process-performance paradox in expert judgment: How can the experts know so much and predict so badly? In K. A. Ericsson and J. Smith, editors, *Towards a general theory of expertise: Prospects and limits*. Cambridge University Press, 1991.

194. J. I. D. Campbell. On the relation between skilled performance of simple division and multiplication. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(5):1140–1159, 1997.

195. J. I. D. Campbell and D. J. Graham. Mental multiplication skill: Structure, process, and acquisition. *Canadian Journal of Psychology*, 39(2):338–366, 1985.

196. J. I. D. Campbell and Q. Xue. Cognitive arithmetic across cultures. *Journal of Experimental Psychology: General*, 130(2):299–315, 2001.

197. B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of WCRE'99, Working Conference on Reverse Engineering*, pages 112–122, Oct. 1999.

198. M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

199. M. Carlo and E. S. Sylvester. Adult second-language reading research: How may it inform assessment and instruction? Technical Report TR96-08, University of Pennsylvania, Oct. 1996.

200. R. E. Carlson, B. H. Khoo, R. G. Yaure, and W. Schneider. Acquisition of a problem-solving skill: Levels of organization and

use of working memory. *Journal of Experimental Psychology: General*, 119(2):193–214, 1990.

201. E. Carmel and S. Becker. A process model for packaged software development. *IEEE Transactions on Engineering Management*, 41(5):50–61, 1995.

202. P. A. Carpenter and M. A. Just. Sentence comprehension: A psycholinguistic processing model of verification. *Psychological Review*, 82(1):45–73, 1975.

203. J. M. Carroll. *What's in a Name? An essay on the psychology of reference*. W. H. Freeman, 1985.

204. A. K. Carter and C. G. Clopper. Prosodic and morphological effects on word reduction in adults: A first report. Technical Report Progress Report No. 24 (2000), Speech Research Laboratory, Indiana University, 2000.

205. L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the* 30^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 106–114, Jan. 2003.

206. R. Carter. Y-MP floating point and Cholesky factorization. Technical Report RND-90-007, NASA Ames Research Center, 1990.

207. E. Caspi. Empirical study of opportunities for bit-level specialization in word-based programs. Thesis (m.s.), University of California, Berkeley, 2000.

208. K. A. Cassell. Tools for the analysis of large PROLOG programs. Thesis (m.s.), University of Texas at Austin, Austin, TX, 1985.

209. R. G. G. Cattell. Automatic derivation of code generators from machine descriptors. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, 1980.

210. J. P. Cavanagh. Relation between the immediate memory span and the memory search rate. *Psychological Review*, 79(6):525–530, 1972.

211. CDC. *6600 Central Processor. Volume II Functional Units*. Control Data Corporation, publication number 60239700 edition, 1966.

212. M. Celce-Murcia, D. M. Brinton, and J. M. Goodwin. *Teaching Pronunciation: A Reference for Teachers of English to Speakers of Other Languages*. Cambridge University Press, 1996.

213. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.

214. S. M. Chambers and K. I. Forster. Evidence for lexical access in a simultaneous matching task. *Memory & Cognition*, 3(5):549–559, 1975.

215. S. Chandra and T. Reps. Physical type checking for C. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engineering Notes (SEN)*, pages 66–75, N. Y., Sept. 6 1999. ACM Press.

216. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software–Practice and Experience*, 22(5):349–369, 1992.

217. W. G. Chase and K. A. Ericsson. Skill and working memory. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, pages 1–58. Academic, 1982.

218. G. Chen. *Effective Instruction Scheduling With Limited Registers*. PhD thesis, Harvard University, Mar. 2001.

219. J. B. Chen. *The Impact of Software Structure and Policy on CPU and Memory System Performance*. PhD thesis, Carnegie Mellon University, May 1994.

220. J. B. Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In USENIX, editor, *The USENIX Windows NT Workshop 1997*, pages 25–32, Berkeley, CA, USA, Aug. 1997. USENIX.

221. W. Y. Chen, P. P. Chang, W. mei W. Hwu, and T. M. Conte. The effect of code expansion optimizations on instruction cache design. Technical Report CRHC-91-17, University of Illinois, Urbana-Champaign, 1991.

222. B.-C. Cheng. *Compile-Time Memory Disambiguation for C Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

223. P. Cheng and K. J. Holyoak. Pragmatic reasoning schemas. *Cognitive Psychology*, 17:391–416, 1985.

224. P. Cheng, K. J. Holyoak, R. E. Nisbett, and L. M. Oliver. Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, 18:293–328, 1986.

225. H. Cheung and S. Kemper. Competing complexity metrics and adults' production of complex sentences. *Applied Psycholinguistics*, 13:53–76, 1992.

226. R. J. Chevance and T. Heidet. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Notices*, 13(4):44–57, Apr. 1978.

227. T. M. Chilimbi. On the stability of temporal data reference profiles. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–162, Sept. 2001.

228. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong. Orthogonal defect classification – A concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, Nov. 1992.

229. D. Chiriaev and G. W. Walster. Interval arithmetic specification. Technical report, Marquette University, May 1998.

230. H.-F. Chitiri and D. M. Willows. Word recognition in two languages and orthographies: English and Greek. *Memory & Cognition*, 22(3):313–325, 1994.

231. Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 182–191. IEEE Computer Society, 2001.

232. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles, SOSP'01*, pages 73–88, 2001.

233. P. Christian. Soundex - can it be improved? *Computers in Genealogy*, 6(5), Dec. 1998.

234. T. W. Christopher. *Icon Programming Language Handbook*. Thomas W. Christopher, 1996.

235. S. Chulani, B. Boehm, and B. Steece. Calibrating software cost models using bayesian analysis. Technical Report USC-CSE-98-508, American Science Institute of Technology, 1998.

236. Z. J. Ciechanowicz and A. C. D. Weever. The 'completeness' of the Pascal test suite. *Software–Practice and Experience*, 14(5):463–471, 1984.

237. M. Ciolkowski, C. Differding, O. Laitenberger, and J. Munch. Empirical investigation of perspective-based reading: a replicated experiment. Technical Report Technical Report ISERN-97-13, Fraunhofer Institute for Experimental Software Engineering, University of Kaiserlautern: Kaiserlautern, 1997.

238. D. Citron. *Instruction Memoization: Exploiting Previously Performed Calculations to Enhance Performance*. PhD thesis, Hebrew University of Jerusalem, Israel, 2000.

239. D. Citron, D. Feitelson, and L. Rudolph. Accelerating multimedia processing by implementing memoing in multiplication and division units. In *Proceedings of* 8^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 252–261, 1998.

240. D. W. Clark and C. C. Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, Feb. 1977.

241. H. H. Clark. Linguistic processes in deductive reasoning. *Psychological Review*, 76(4):387–404, 1969.

242. H. H. Clark. *Understanding language*. Cambridge University Press, 1996.

243. H. H. Clark and W. G. Chase. On the process of comparing sentences against pictures. *Cognitive Psychology*, 3:472–517, 1972.

244. H. H. Clark and D. Wilkes-Gibbs. Referring as a collaborative process. *Cognition*, 22:1–39, 1986.

245. J. J. Clark and J. K. O'Regan. Word ambiguity and the optimal viewing position in reading. *Vision Research*, 39(4):843–857, 1998.

246. R. L. Clark. A linguistic contribution of GOTO-less programming. *Datamation*, 19(12):62–63, Dec. 1973.

247. W. D. Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101, June 1990.

248. R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 290–301, 1991.

249. W. J. Cody Jr. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.

250. G. Cohen. Why is it difficult to put names to faces? *British Journal of Psychology*, 81:287–297, 1990.

251. G. Cohen and D. Faulkner. Memory for proper names: Age differences in retrieval. *British Journal of Psychology*, 4:187–197, 1986.

252. R. Cohn and P. G. Lowney. Design and analysis of profile-based optimization in Compaq's compilation tools for alpha. *Journal of Instruction-Level Parallelism*, 3:1–25, 2000.

253. J. Coleman and J. Pierrehumbert. Stochastic phonological grammars and acceptability. In *Computational phonology: Third meeting of the ACL special interest group in computational phonology*, pages 49–56. Association for the Computational Linguistics, 1997.

254. J. N. Coleman and E. I. Chester. A 32-bit logarithmic arithmetic unit and its performance compared to floating-point. In *Proceedings of the* 14^{th} *Symposium on Computer Arithmetic*, pages 142–151, 1999.

255. J. N. Coleman, C. I. Softley, J. Kadlec, R. Matoušek, Z. Pohl, and A. Heřmáek. The european logarithmic microprocessor - a QR RLS application. Technical Report No. 2038, Institute of Information Theory and Automation, Czech Republic, Dec. 2001.

256. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report Technical Report 148, Department of Computer Science, University of Auckland, 1997.

257. Collins. *Advanced Learner's English Dictionary*. Collins COBUILD, 2003.

258. A. M. Collins and M. R. Quillian. Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247, 1969.

259. R. Colom, I. Rebollo, A. Palacios, M. Juan-Espinosa, and P. C. Kyllonen. Working memory is (almost) perfectly predicted by g. *Intelligence*, 32:277–296, 2004.

260. M. Coltheart, K. Rastle, C. Perry, R. Langdon, and J. Ziegler. DRC: A dual route cascaded model of visual word recognition and reading aloud. *Psychological Review*, 108(1):204–256, 2001.

261. V. Coltheart. Effects of phonological similarity and concurrent irrelevant articulation on short-term-memory recall of repeated and novel word lists. *Memory & Cognition*, 21(4):539–545, 1993.

262. A. Colvin. *VIC* running out-of-core instead of running out of core*. PhD thesis, Dartmouth College, June 1999.

263. A. Computer. *MrC/MrCpp C/C++ Compiler for Power Macintosh*. Apple Computer, Inc, 1.0 edition, 1995.

264. B. Comrie. Conditionals: A typology. In E. C. Traugott, A. T. Meulen, J. S. Reilly, and C. A. Furguson, editors, *On Conditionals*, chapter 4, pages 77–97. Cambridge University Press, 1986.

265. B. Comrie. *Language Universals and Linguistic Typology*. Blackwell, second edition, 1989.

266. G. K. Connolly. Legibility and readability of small print: Effects of font, observer age and spatial vision. Thesis (m.s.), University of Calgary, Alberta, Canada, Feb. 1998.

267. D. A. Connors, Y. Yamada, and W. mei W. Hwu. A software-oriented floating-point format for automotive control systems. In *Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES'98)*, 1998.

268. R. Conrad. Acoustic confusions in immediate memory. *British Journal of Psychology*, 55(1):75–84, 1964.

269. R. Conrad. Order error in immediate recall of sequences. *Journal of Verbal Learning and Verbal Behavior*, 4:161–169, 1965.

270. S. P. Consortium. Ada 95 quality and style guide: Guidelines for professional programmers. Technical Report SPC-94093-CMC Version 01.00.10, Software Productivity Consortium, Oct. 1995.

271. D. Conway. *Perl Best Practices*. O'Reilly, 2005.

272. V. Cook. *Second Language Learning and Language Teaching*. Arnold, third edition, 2001.

273. V. J. Cook. *Inside Language*. Arnold, 1997.

274. V. J. Cook. L2 users and English spelling. *Journal of Multilingual and Multicultural Development*, 18(6):474–488, 1997.

275. K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, Mar. 1992.

276. K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

277. K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, Sept. 2001.

278. K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. In *Workshop on Memory Systems Performance (MSP '02)*, pages 97–107. ACM Press, June 2002.

279. J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison–Wesley, 1991.

280. R. Coppieters. Competence differences between native and near-native speakers. *Language*, 63(3):544–573, 1987.

281. G. G. Corbett and I. R. L. Davies. Establishing basic color terms: measures and techniques. In C. L. Hardin and L. Maffi, editors, *Color categories in thought and language*, chapter 9, pages 197–223. Cambridge University Press, 1997.

282. R. P. Corbett. *Static Semantics and Compiler Error Recovery*. PhD thesis, University of California, Berkeley, June 1985. Report No. UCB/CSD 85/251.

283. D. W. J. Corcoran. Acoustic factor in proof reading. *Nature*, 214:851–852, May 1967.

284. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.

285. L. Cosmides. The logic of social exchange: Has natural selection shaped how humans reason? Studies with the Wason selection task. *Cognition*, 31:187–276, 1989.

286. L. Cosmides and J. Tooby. Evolutionary psychology: A primer. Technical report, Center for Evolutionary Psychology, University of California, Santa Barbara, 1998.

287. F. Costello and M. T. Keane. Polysemy in conceptual combination: Testing the constraint theory of combination. In P. L. M. G. Shafto, editor, *Nineteenth Annual Conference of the Cognitive Science Society*, pages 137–142. Erlbaum, Apr. 1997.

288. F. J. Costello. Efficient creativity: Constraint-guided conceptual combination. *Cognitive Science*, 24(2):299–349, 2000.

289. F. J. Costello. Testing a computational model of categorisation and category combination: Identifying disease categories and new disease combinations. In J. D. Moore and K. Stenning, editors, *Proceedings of the Twenty-Third Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates, Aug. 2001.

290. F. J. Costello and M. T. Keane. Testing two theories of conceptual combination: Alignment versus diagnosticity in the comprehension and production of combined concepts. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 27(1):255–271, 2001.

291. J. D. Couger and M. A. Colter. *Maintenance Programming: Improving Productivity Through Motivation*. Prentice-Hall, Inc, 1985.

292. N. S. Coulter and N. H. Kelly. Computer instruction set usage by programmers: An empirical investigation. *Communications of the ACM*, 29(7):643–647, July 1986.

293. M. A. Covington. Some coding guidelines for Prolog. www.ai.uga.edu/mc, 2001.

294. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7^{th} USENIX Security Conference*, pages 63–78, Jan. 1998.

295. N. Cowan. *Attention and Memory: An Integrated Framework*. Oxford University Press, 1997.

296. N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–185, 2001.

297. M. Cowlishaw. General decimal arithmetic specification. Technical Report Version 1.30, IBM UK Laboratories, June 2003.

298. M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In *16^{th} IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, pages 104–111, June 2003.

299. Cray. *Cray-1 Computer Systems: Cray-1 S Series Hardware Reference Manual*. Cray Research, Inc, Nov. 1981.

300. E. R. F. W. Crossman. A theory of the acquisition of speed-skill. *Ergonomics*, 2:153–166, 1959.

301. A. Cruttenden. *Gimson's Pronunciation of English*. Arnold, sixth edition, 2001.

302. M. Csikszenthmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, 1990.

303. A. Cutler. The reliability of speech error data. In A. Cutler, editor, *Slips of the tongue and language production*, chapter Guest editorial, pages 7–28. Mouton, 1982.

304. A. Cutler and D. M. Carter. The predominance of strong initial syllables in the English vocabulary. *Computer Speech and Language*, 2:133–142, 1987.

305. A. Cuyt, P. Kuterna, B. Verdonk, and D. Verschaeren. Underflow revisited. *Calcolo*, 39(3):169–179, 2002.

306. A. Cuyt and B. Verdonk. A remarkable example of catastrophic cancellation unraveled. *Computing*, 66(3):309–320, 2001.

307. W. Daelemans and A. van den Bosch. Language-independent data-oriented grapheme-to-phoneme conversion. In J. P. H. van Santen, R. W. Sproat, J. P. Olive, and J. Hirschberg, editors, *Progress in Speech Synthesis*, pages 77–89. Springer, New York, 1997.

308. B. Daille, B. Habert, C. Jacquemin, and J. Royauté. Empirical observation of term variations and principles for their description. *Terminology*, 3(2):197–258, May 1996.

309. R. Dale and E. Reiter. Computational interpretations of the Gricean maxims in the generating referring expression. *Cognitive Science*, 19(2):233–263, 1995.

310. R. Dallaway. Dynamics of arithmetic connectionist view of arithmetic skills. Technical Report CSRP 306, University of Sussex, 1994.

311. F. J. Damerau and E. Mays. An examination of undetected typing errors. *Information Processing & Management*, 25(6):659–664, 1989.

312. R. I. Damper, Y. Marchand, M. J. Adamson, and K. Gustafson. Evaluating the pronunciation component of text-to-speech systems for English: A performance comparison of different approaches. *Computer Speech and Language*, 13(2):155–176, 1999.

313. M. Daneman and P. A. Carpenter. Individual differences in working memory and reading. *Journal of Verbal Learning and Verbal Behavior*, 19:450–466, 1980.

314. M. Daneman and P. A. Carpenter. Individual differences in integrating information between and within sentences. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 9(4):561–584, 1983.

315. M. Daneman and M. Stainton. Phonological recoding in silent reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(4):618–632, 1991.

316. J. D. Darcy and D. Gay. FLECKmarks measuring floating point performance using a FulL IEEE Compliant arithmetic BenchmarK. Technical report, U.C. Berkeley, Aug. 1996.

317. J. Darley and C. D. Batson. From Jerusalem to Jericho: A study of situational and dispositional variables in helping behavior. *Journal of Personality and Social Psychology*, 27(1):100–108, 1973.

318. M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, volume 35.5 of *ACM SIGPLAN Notices*, pages 35–46, N.Y., June 18–21 2000. ACM Press.

319. Data General Corporation. *Programmer's Reference Manual: Nova Line Computers*. Data General Corporation, 2 edition, Sept. 1975.

320. R. Dattero and S. D. Galup. Programming languages and gender. *Communications of the ACM*, 47(1):99–102, Jan. 2004.

321. T. H. Davenport and J. C. Beck. *The Attention Economy*. Harvard Business School Press, 2001.

322. J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software–Practice and Experience*, 18(8):775–790, 1988.

323. J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.

324. J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical Report CS-95-26, University of Virginia, June 1995.

325. J. W. Davidson, J. R. Rabung, and D. B. Whalley. Relating static and dynamic machine code measurements. Technical Report CS-89-03, Department of Computer Science, University of Virginia, July 13 1989.

326. J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimization. *Software–Practice and Experience*, 19(1):79–97, 1989.

327. J. W. Davidson and D. B. Whalley. Methods for saving and restoring register values across function calls. *Software–Practice and Experience*, 21(2):459–472, Feb. 1991.

328. C. J. Davies and J. S. Bowers. Contrasting five different theories of letter position coding: Evidence from orthographic similarity effects. *Journal of Experimental Psychology: Human Perception and Performance*, 32(3):535–557, 2006.

329. M. Davis. Text boundaries. Technical Report 29, The Unicode Consortium, Mar. 2005.

330. J. W. Davison, D. M. Mand, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, Apr.-June 2000.

331. G. C. S. de Araújo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, 1997.

332. R. W. De Lange, H. L. Esterhuizen, and D. Beatty. Performance differences between times and helvetica in a reading task. *Electronic Publishing*, 6(3):241–248, Sept. 1993.

333. C. B. De Soto, M. London, and S. Handel. Social reasoning and spatial paralogic. *Journal of Personality and Social Psychology*, 2(4):513–521, 1965.

334. T. W. Deacon. *The Symbolic Species: The Co-evolution of Language and the Brain*. W. W. Norton, 1997.

335. I. J. Deary and C. Stough. Intelligence and inspection time. *American Psychologist*, 51(6):599–608, 1996.

336. A. Degani and E. L. Wiener. On the design of flight-deck procedures. Technical Report 177642, NASA Ames Research Center, June 1994.

337. P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software–Practice and Experience*, 25(6):657–679, 1995.

338. S. Dehaene. *The Number Sense*. Penguin, 1997.

339. S. Dehaene and R. Akhavein. Attention, automaticity, and levels of representation in number processing. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(2):314–326, 1995.

340. S. Dehaene, E. Dupoux, and J. Mehler. Is numerical comparison digits? Analogical and symbolic effects in two-digit number comparisons. *Journal of Experimental Psychology: Human Perception and Performance*, 16(3):626–641, 1990.

341. L. E. Deimel and J. F. Naveda. Reading computer programs: Instructor's guide and exercises. Technical Report CMU/SEI-90-EM-3 ADA228026, Software Engineering Institute (Carnegie Mellon University), 1990.

342. B. L. Deitrich. *Static Program Analysis to Enhance Profile Independence in Instruction-Level Parallelism Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.

343. M. Delazer and T. Benke. Arithmetic facts without meaning. *Cortex*, 33:697–710, 1997.

344. J. Demmel and Y. Hilda. Accurate floating-point summation. Technical Report UCB//CSD-02-1180, University of California, Berkeley, May 2002.

345. B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. In *ICLP Workshop: Parallel and Data Parallel Execution of Logic Programs*, pages 79–91, 1994.

346. P. Desberg, D. E. Elliott, and G. Marsh. American black English spelling. In U. Frith, editor, *Cognitive Processes in Spelling*, chapter 4, pages 69–82. Academic Press, 1980.

347. N. Deshmukh. *Maximum likelihood estimation of multiple pronunciations for proper names*. PhD thesis, Mississippi State University, June 1999.

348. P. Deshpande and A. Somani. A study and analysis of function inlining. www.cs.wisc.edu/~pmd/pmd.html, 199?

349. D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Aug. 1993.

350. A. M. Devices. *3DNow! Technology Manual*. Advanced Micro Devices, Inc, 2000.

351. D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the 28th International Conference on Software Engineering*, pages 162–171, 2006.

352. B. Di Martino. Specification and automatic recognition of algorithmic concepts within programs. Technical Report Technical Report 97-14, University of Vienna, Nov. 1997.

353. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.

354. L. S. Dickstein. The effect of figure on syllogistic reasoning. *Memory & Cognition*, 6(1):76–83, 1978.

355. H. G. Dietz and T. I. Mattox. Compiler optimizations using data compression to decrease address reference entropy. In *LCPC '02: 15^{th} Workshop on Languages and Compilers for Parallel Computing*, July 2002.

356. Digitalmars. Digitalmars C compiler. www.digitalmars.com, 2003.

357. E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

358. A. Dillon. Reading from paper versus screens: a critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, 1992.

359. C. DiMarco, G. Hirst, and M. Stede. The semantic and stylistic differentiation of synonyms and near-synonyms. In *AAAI Spring Symposium on Building Lexicons for Machine Translation*, pages 114–121, Mar. 1993.

360. D. K. Dirlam. Most efficient chunk sizes. *Cognitive Psychology*, 3:355–359, 1972.

361. R. Dirven. Dividing up physical and mental space into conceptual categories by means of English prepositions. In C. Zelinksy-Wibbelt, editor, *Natural Language processing (vol. 3, The Semantics of Prepositions)*, pages 73–97. Mouton de Gruyter, 1993.

362. D. R. Ditzel and A. D. Berenbaum. Design tradeoffs to support the C programming language in the CRISP microprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems-ASPLOSII*, pages 158–163. IEEE Computer Society Press, Oct. 1987.

363. D. R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proceedings of the First International Symposium on Architectural support for Programming Languages and Operating Systems*, pages 48–56, 1982.

364. M. Divay and T. Vitale. Algorithms for grapheme-phoneme translation for English and French: Applications for database searches and speech synthesis. *Computational Linguistics*, 23(4):496–523, 1997.

365. K. M. Dixit. Overview of the SPEC benchmarks. In J. Gray, editor, *The Benchmark Handbook*, chapter 9, pages 489–521. Morgan Kaufmann Publishers, 1993.

366. D. J. Dooling and R. E. Christiaansen. Episodic and semantic aspects of memory for prose. *Journal of Experimental Psychology: Human Learning and Memory*, 3(4):428–436, 1977.

367. S. N. Dorogovtsev, J. F. F. Mendes, and J. G. Oliveira. Frequency of occurrence of numbers in the World Wide Web. *Physica A*, 360(2):548–556, 2006.

368. P. Downing. On the creation and use of English compound nouns. *Language*, 53(4):810–842, 1977.

369. A. Drewnowski and A. F. Healy. Phonetic factors in letter detection: A reevaluation. *Memory & Cognition*, 10(2):145–154, 1982.

370. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicate code. In H. Yang and L. White, editors, *Proceedings International Conference on Software Maintenance ICSM'99*, pages 109–119. IEEE Computer Society Press, Sept. 1999.

371. T. Duff. Unwinding loops. *Newsgroup: netlang.c*, 1984-05-07 07:19:21 PST.

372. A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the Digital Aviation Systems Conference (DASC)*, pages 1B6/1–1B6/7, Oct. 2000.

373. T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 2006.

374. H. Ebbinghaus. *Memory: A contribution to experimental psychology*. Dover Publications, 1987.

375. E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 99)*, volume 34.7 of *ACM SIGPLAN Notices*, pages 20–27. ACM Press, May 5 1999.

376. B. Edelman, H. Abdi, and D. Valentin. Multiplication number facts: Modeling human performance with connectionist networks. *Psychologica Belgica*, 36(1/2):31–63, Apr. 1996.

377. P. Edmonds. *Semantic Representation of Near-Synonyms for Automatic Lexical Choice*. PhD thesis, University of Toronto, 1999.

378. L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. How input data sets change program behaviour. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW'02)*, Feb. 2002.

379. S. L. Ehrenreich and T. Porcu. Abbreviations for automated systems: Teaching operators the rules. In A. Badre and B. Shneiderman, editors, *Directions in Human/Computer Interaction*, chapter 6, pages 111–135. Ablex Publishing Corp., 1982.

380. K. Ehrlich and P. N. Johnson-Laird. Spatial descriptions and referential continuity. *Journal of Verbal Learning and Verbal Behavior*, 21:296–306, 1982.

381. W. H. Eichelman. Familiarity effects in the simultaneous matching task. *Journal of Experimental Psychology*, 86(2):275–282, 1970.

382. S. G. Eick, T. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

383. H. J. Einhorn. Accepting error to make less error. *Journal of Personality Assessment*, 50:387–395, 1986.

384. N. C. Ellis and R. A. Hennelly. A bilingual word-length effect: Implications for intelligence testing and the relative ease of mental calculation in Welsh and English. *British Journal of Psychology*, 71:43–51, 1980.

385. R. Ellis and G. Humphreys. *Connectionist Psychology*. Psychology Press, 1999.

386. J. Ellman. *Using Roget's Thesaurus to Determine Similarity of Texts*. PhD thesis, University of Sunderland, June 2000.

387. J. L. Elshoff. A numerical profile of commercial PL/I programs. *Software–Practice and Experience*, 6:505–525, 1976.

388. K. E. Emam and I. Wieczorek. The repeatability of code defect classifications. Technical Report Technical Report ISERN-98-09, Fraunhofer Institute for Experimental Software Engineering, 1998.

389. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.

390. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.

391. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Apr. 2002.

392. J. Epelboim, J. R. Booth, R. Ashkenazy, A. Taleghani, and R. M. Steinmans. Fillers and spaces in text: The importance of word recognition during reading. *Vision Research*, 37(20):2899–2914, 1997.

393. K. A. Ericsson and N. Charness. Expert performance. *American Psychologist*, 49(8):725–747, 1994.

394. K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100:363–406, 1993. also University of Colorado, Technical Report #91-06.

395. K. A. Ericsson and A. C. Lehmann. Expert and exceptional performance: Evidence of maximal adaption to task constraints. *Annual Review of Psychology*, 47:273–305, 1996.

396. M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

397. A. M. Erosa. A goto-elimination method and its implementation for the McCat C compiler. Thesis (m.s.), McGill University, Montreal, Canada, May 1995.

398. W. K. Estes. *Classification and Cognition*. Oxford University Press, 1994.

399. L. H. Etzkorn, L. L. Bowen, and C. G. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(1):1–18, 1999.

400. D. Evans and D. Larochelle. *Splint Manual*. University of Virginia, 3.0.6 edition, Feb. 2002.

401. J. S. B. T. Evans. Deciding before you think: Relevance and reasoning in the selection task. *British Journal of Psychology*, 87:223–240, 1996.

402. J. S. B. T. Evans, J. L. Barston, and P. Pollard. On the conflict between logic and belief in syllogistic reasoning. *Memory & Cognition*, 11(3):295–306, 1983.

403. J. S. B. T. Evans and S. E. Newstead. A study of disjunctive reasoning. *Psychological Research*, 41(4):373–388, Apr. 1980.

404. K. Ewusi-Mensah and Z. H. Przasnyski. On information systems project abandonment: An exploratory study of organizational practices. *MIS Quarterly*, 15(1):67–86, Mar. 1991.

405. M. W. Eysenck and M. T. Keane. *Cognitive Psychology: A Student's Handbook*. Psychology Press, fourth edition, 2000.

406. C. Fagot. *Chronometric investigations of task switching*. PhD thesis, University of California, San Diego, 1994.

407. R. Falk and C. Konold. Making sense of randomness: Implicit encoding as a basis for judgment. *Psychological Review*, 104(2):301–318, 1997.

408. M. Farach and V. Liberatore. On local register allocation. Technical Report DIMACS Technical Report 97-33, Rutgers University, July 1997.

409. S. Farrell and S. Lewandowsky. Dissimilar items benefit from phonological similarity in serial recall: The dissimilar immunity effect reviseted. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 29(5):838–849, 2003.

410. D. Fasulo. An analysis of recent work on clustering algorithms. Technical Report Technical Reptort 01-03-02, Dept. of Computer Science and Engineering, University of Washington, 1999.

411. R. J. Fateman. Software fault prevention by language choice: Why C is not my favorite language. University of California at Berkeley, 1999.

412. J. M. Favaro. Value based software reuse investment. *Annals of Software Engineering*, 5:5–52, 1998.

413. D. Fay and A. Cutler. Malapropisms and the structure of the mental lexicon. *Linguistic Inquiry*, 8(3):505–520, 1977.

414. J. Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407:630–633, Oct. 2000.

415. A. Feldstein and P. Turner. Overflow, underflow, and severe loss of significance in floating-point addition and subtraction. *IMA Journal of Numerical Analysis*, 6:241–251, 1986.

416. C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

417. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(3):675–689, 1999.

418. N. E. Fenton and M. Neil. Software metrics: Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 357–370. ACM Press, 2000.

419. N. E. Fenton and S. L. Pfleeger. *Software Metrics*. PWS Publishing Company, second edition, 1997.

420. S. Feuerstein. *Oracle PL/SQL Best Practices*. O'Reilly, 2001.

421. K. Fiedler. The dependence of the conjunction fallacy on subtle linguistic factors. *Psychological Research*, 50:123–129, 1988.

422. S. A. Figueroa del Cid. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, New York University, Jan. 2000.

423. C. J. Fillmore. Topics in lexical semantics. In R. W. Cole, editor, *Current Issues in Linguistic Theory*, pages 76–138. Indiana University Press, 1977.

424. K. Finney and N. Fenton. Evaluating the effectiveness of Z: the claims made about CICS and where do we go from here. Technical Report DeVa TR No. 05, City University, London, 1996.

425. E. Fischer. The evolution of character codes, 1874-1968. Nov. 2002.

426. D. L. Fisher. Central capacity limits in consistent mapping, visual search tasks: Four channels or more? *Cognitive Psychology*, 16:449–484, 1984.

427. J. E. Fisk and C. Sharp. Syllogistic reasoning and cognitive ageing. *The Quarterly Journal of Experimental Psychology*, 55A(4):1273–1293, 2002.

428. G. J. Fitzsimons and B. Shiv. Non-conscious and contaminative effects of hypothetical questions on subsequent decision making. *Journal of Consumer Research*, 28:224–238, Sept. 2001.

429. R. K. Fjelstad and W. T. Hamlen. Applications program maintenance study: Report to our respondents. In G. Parikh and N. Zvegintzov, editors, *Tutorial on Software Maintenance*. IEEE Computer Society Press, 1979.

430. A. D. Flatau. A Pascal to Lisp translator for the Symbolics 3600 Lisp machine. Thesis (m.s.), University of Texas at Austin, Austin, TX, 1985.

431. J. H. Flowers and D. J. Lohr. How does familiarity affect visual search for letter strings? *Perception and Psychophysics*, 37:557–567, 1985.

432. B. Fluri, M. Würsch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE)*, page ???, Oct. 2007.

433. J. R. Folk. Phonological codes are used to access the lexicon during silent reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(4):892–906, 1999.

434. B. Foote and J. Yoder. Big ball of mud. In *Fourth Conference on Pattern Languages of Programs (PLoP) 1997*, 1997.

435. C. E. Ford and S. A. Thompson. Conditionals in discourse: A text-based study from English. In E. C. Traugott, A. T. Meulen, J. S.

Reilly, and C. A. Furguson, editors, *On Conditionals*, chapter 18, pages 353–372. Cambridge University Press, 1986.

436. J. S. Foster and et al. CQUAL *User's Guide*. University of California, Berkeley, version 0.9 edition, Jan. 2002.

437. J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 192–203, 1999.

438. J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):1035–1087, Nov. 2006.

439. K. J. Fowler. A study of implementation-dependent pragmas and attributes in Ada. Technical Report CMU-SEI-89-SR-19, Software Engineering Institute, Carnegie Mellon University, Nov. 1989.

440. W. B. Frakes, C. J. Fox, and B. A. Nejmeh. *Software Engineering in the Unix/C Environment*. Prentice Hall, Inc, 1991.

441. B. Franke and M. O'Boyle. Compiler transformation of pointers to explicit array accesses in DSP applications. In R. Wilhelm, editor, *Compiler Construction,* 10^{th} *International Conference*, pages 69–85. Springer-Verlag, Apr. 2001.

442. M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the* 25^{th} *Annual International Symposium on Microarchitecture (MICRO-25)*, pages 236–245, 1992.

443. M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In 10^{th} *USENIX Security Symposium*, pages 55–66, Aug. 2001.

444. M. Franz and T. Kistler. Splitting data objects to increase cache utilization. Technical Report Technical Report No. 98-34, Department of Information and Computer Science, University of California, Irvine, 1998.

445. M. Frappier, S. Matwin, and A. Mili. Software metrics for predicting maintainability. Technical Report Technical Memorandum 2, Canadian Space Agency, Jan. 1994.

446. L. T. Frase. Associative factors in syllogistic reasoning. *Journal of Experimental Psychology*, 76(3):407–412, 1968.

447. C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

448. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

449. U. H. Frauenfelder, R. H. Baayen, F. M. Hellwig, and R. Schreuder. Neighborhood density and frequency across languages and modalities. *Journal of Memory and Language*, 32:781–804, 1993.

450. S. Frederick, G. Loewenstein, and T. O'Donoghue. Time discounting: A critical review. *Journal of Economic Literature*, 40(2):351–401, 2002.

451. M. Fredericks. Using defect tracking and analysis to improve software quality. Thesis (m.s.), Department of Computer Science, University of Maryland, 1999.

452. D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House Publishing, 1990.

453. C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1):199–227, 1992.

454. S. Frisch. *Similarity and frequency in phonology*. PhD thesis, Northwestern University, Dec. 1996.

455. S. A. Frisch, N. R. Large, and D. B. Pisoni. Perception of word-likeness: Effects of segment probability and length of the processing of nonwords. *Journal of Memory and Language*, 42:481–496, 2000.

456. R. Frost. Phonological computations and missing vowels: Mapping lexical involvement in reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(2):398–408, 1995.

457. R. Frost, L. Katz, and S. Bentin. Strategies for visual word recognition and orthographical depth: A multilingual comparison. *Journal of Experimental Psychology: Human Perception and Performance*, 13(1):104–115, 1987.

458. W.-T. Fu and W. D. Gray. Memory versus perceptual-motor tradeoffs in a blocks world task. In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pages 154–159, Hillsdale, NJ, 2000. Erlbaum.

459. E. Fudge. *English Word-Stress*. George Allen & Unwin, 1984.

460. M. Fuller and J. Zobel. Conflation-based comparison of stemming algorithms. In *Proceedings of the Third Australian Document Computing Symposium*, pages 8–13, Aug. 1998.

461. B. Furht. A RISC architecture with two-size, overlapping register windows. *IEEE Micro*, 8(2):67–80, Mar. 1988.

462. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. Statistical semantics: Analysis of the potential performance of key-word information systems. *The Bell System Technical Journal*, 62(6):1753–1805, 1983.

463. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication: an analysis and a solution. *Communications of the ACM*, 30(11):964–971, 1987.

464. T. Furugori. Improving spelling checkers for Japanese users of English. *IEEE Transactions on Professional Communication*, 33(3):138–142, Sept. 1990.

465. X. Gabaix. Zipf's law for cities: An explanation. *The Quarterly Journal of Economics*, 114(4):739–767, Aug. 1999.

466. F. Gabbay. Speculative execution based on value prediction. Technical Report EE Department technical Report #1080, Technion - Israel Institute of Technology, Nov. 1996.

467. F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 270–280. IEEE, Dec. 1997.

468. A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 523–533. Springer-Verlag, Apr. 2002.

469. H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM'97*, pages 160–166, 1997.

470. R. Ganesan and A. T. Sherman. Statistical techniques for language recognition: An introduction and guide for cryptanalysts. *Cryptologia*, XVII(4):321–366, Oct. 1993.

471. H. Gardner. *Intelligence Reframed*. Basic Books, 1999.

472. M. K. Gardner, E. Z. Rothkopf, R. Lapan, and T. Laferty. The word frequency effect in lexical decision: Finding a frequency-based component. *Memory & Cognition*, 15(1):24–28, 1987.

473. S. Garrod and G. Doherty. Conversation, co-ordination and convention: an empirical investigation of how groups establish linguistic conventions. *Cognition*, 53:181–215, 1994.

474. S. E. Gathercole, S. J. Pickering, C. Knight, and Z. Stegmann. Working memory skills and education attainment: Evidence form national curriculum assessments at 7 and 14 years of age. *Applied Cognitive Psychology*, 18(1):1–16, 2004.

475. D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, Murray Hill, NJ, USA, Nov. 1990.

476. N. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.

477. N. H. Gehani. Exceptional C or C with exceptions. *Software–Practice and Experience*, 22(10):827–848, 1992.

478. E. M. Gellenbeck and C. R. Cook. Does signaling help professional programmers read and understand computer programs? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 82–98, 1991.

479. W. Gellerich, M. Kosiol, and E. Ploedereder. Where does GOTO go to? In *Reliable Software Technology —Ada-Europe 1996*, volume 1088 of *LNCS*, pages 385–395. Springer, 1996.

480. S. A. Gelman and E. M. Markman. Categories and induction in young children. *Cognition*, 23:183–209, 1986.

481. D. R. Gentner, S. Larochelle, and J. Grudin. Lexical, sublexical, and peripheral effects in skilled typewriting. *Cognitive Psychology*, 20:524–548, 1988.

482. M. J. Gervais, L. O. Harvey, and J. O. Roberts. Identification confusions among letters of the alphabet. *Journal of Experimental Psychology: Human Perception and Performance*, 10(5):655–666, 1984.

483. R. A. Ghosh, R. Glott, B. Krieger, and G. Robles. Free/Libre and open source software survey and study, part 4: Survey of developers. Technical Report Deliverable D18: Final Report, University of Maastricht, June 2002.

484. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

485. T. H. Gibbs. *The design and implementation of a parser and front-end for the ISO C++ language and validation of the parser*. PhD thesis, Clemson University, May 2003.

486. E. Gibson and J. Thomas. Memory limitations and structured forgetting: The perception of complex ungrammatical sentences as grammatical. *Language and Cognitive Processes*, 14(3):225–248, 1999.

487. E. J. Gibson, A. Pick, H. Osser, and M. Hammond. The role of grapheme-phoneme correspondence in the perception of words. *American Journal of Psychology*, 75:554–570, 1962.

488. A. Gierlinger, R. Forsyth, and E. Ofner. GEPARD: A parameterisable DSP core for ASICS. In *Proceedings ICSPAT'97*, pages 203–207, 1997.

489. G. Gigerenzer, P. M. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999.

490. D. T. Gilbert, R. W. Tafarodi, and P. S. Malone. You can't not believe everything you read. *Journal of Personality and Social Psychology*, 65(2):221–233, 1993.

491. A. S. Gilinsky and B. B. Judd. Working memory and bias in reasoning across the life span. *Psychology and Ageing*, 9(3):356–371, 1994.

492. G. C. Gilmore, H. Hersh, A. Caramazza, and J. Griffin. Multidimensional letter similarity derived from recognition errors. *Perception & Psychophysics*, 25(5):425–431, 1979.

493. Gimpel Software. *Reference Manual for PC-lint*. Gimpel Software, 6.00 edition, Jan. 1994.

494. V. Girotto, A. Mazzocco, and A. Tasso. The effect of premise order on conditional reasoning: a test of the mental model theory. *Cognition*, 63:1–28, 1997.

495. M. Glanzer, B. Fischer, and D. Dorfman. Short-term storage in reading. *Journal of Verbal Learning and Verbal Behavior*, 23:467–486, 1984.

496. R. L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, 7(2):162–168, Mar. 1981.

497. A. M. Glenberg and W. Epstein. Inexpert calibration of comprehension. *Memory & Cognition*, 15(1):84–93, 1987.

498. D. R. Godden and A. D. Baddeley. Context-dependent memory in two natural environments: On land and underwater. *British Journal of Psychology*, 66(3):325–331, 1975.

499. M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 131–142, Oct. 2000.

500. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.

501. R. A. Goldberg, S. Schwartz, and M. Stewart. Individual differences in cognitive processes. *Journal of Educational Psychology*, 69(1):9–14, 1977.

502. A. R. Golding and P. S. Rosenbloom. A comparison of Anapron with seven other name-pronunciation systems. Technical Report TR-93-05a, Mitsubishi Electric Research Laboratories, May 1996.

503. J. M. Goldschneider and R. M. DeKeyser. Explaining the "Natural order of L2 morpheme acquisition" in English: A meta-analysis of multiple determinants. *Language Learning*, 51(1):1–50, 2001.

504. P. F. D. Gontijo, J. Rayman, S. Zhang, and E. Zaidel. How brand names are special: brands, words, and hemispheres. *Brain and Language*, 82:327–343, 2002.

505. E. G. Gonzalaz and P. A. Kolers. Mental manipulation of arithmetic symbols. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8(4):308–319, 1982.

506. H. Goodglass and A. Wingfield. Selective preservation of a lexical category in aphasia: Disassociations in comprehension of body parts and geographical places names following focal brain lesion. *Memory*, 1(4):313–328, 1993.

507. J. Gosling. Ace: a syntax-driven C preprocessor. In *Australian Unix Users Group*, July 1989.

508. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.

509. S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165. IEEE Press, May 2003.

510. A. C. Graesser, N. L. Hoffman, and L. F. Clark. Structural components of reading time. *Journal of Verbal Learning and Verbal Behavior*, 19:135–151, 1980.

511. A. C. Graesser, S. B. Woll, D. J. Kowalski, and D. A. Smith. Memory for typical and atypical actions in scripted activities. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):503–515, 1980.

512. J. Grainger, J. K. O'Regan, A. M. Jacobs, and J. Segui. Neighborhood frequency effects and letter visibility in visual word recognition. *Perception & Psychophysics*, 51(1):49–56, 1992.

513. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, Mar. 1997.

514. E. E. Grant and H. Sackman. An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, 8(1):33–48, Mar. 1967.

515. S. Graphics. *Cray Standard C/C++ Reference Manual*. Silicon Graphics, Inc, Mountain View, CA, USA, 3.6 edition, June 2002.

516. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

517. J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In L. Wills, I. Baxter, and E. Chikofsky, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 227–238. IEEE Computer Society Press, Nov. 1996.

518. D. Green and P. Meara. The effects of script on visual search. *Second Language Research*, 3(2):102–118, 1987.

519. D. W. Green, E. J. Hammond, and S. Supramaniam. Letters and shapes: Developmental changes in search strategies. *British Journal of Psychology*, 74:11–16, 1983.

520. P. Grice. *Studies in the Way of Words*. Harvard University Press, 1989.

521. R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL 4 Programming Language*. Prentice Hall, Inc, second edition, 1968.

522. A. J. M. Groenewegen and W. A. Waganaar. Diagnosis in everyday situations: Limitations of performance at the knowledge level. Unit of Experimental Psychology, Leiden University, 1988.

523. W. Groop. Users manual for doctext: Producing documentation from C source code. Technical Report Technical Report ANL/MCS-TM-206, Argonne National Laboratory, University of Chicago, Mathematics and Computer Science Division, 1995.

524. D. Grune, H. E. Bel, C. J. H. Jacobs, and K. G. Langerndoen. *Modern Compiler Design*. John Wiley & Sons, Ltd, 2000.

525. D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

526. D. Grunwald and B. Zorn. CUSTOMALLOC: Efficient synthesized memory allocators. Technical Report CU-CS-602-92, University of Colorado at Boulder, July 1992.

527. R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimizations. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 4, pages 143–174. CRC Press, 2002.

528. Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Boerger, H. K. Buning, G. Jager, S. Martini, and M. M. Richter, editors, *Computer Science Logic: Selected Papers from CSL'92*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

529. P. Gutmann. Verification techniques. In P. Gutmann, editor, *Design and Verification of a Cryptographic Security Architecture*, chapter 4. Springer-Verlag, 2003.

530. H. Haarmann and M. Usher. Maintenance of semantic information in capacity-limited short-term memory. *Psychonomic Bulletin & Review*, 8(3):568–578, 2001.

531. R. N. Haber and R. M. Schindler. Error in proofreading: Evidence of syntactic control of letter processing? *Journal of Experimental Psychology: Human Perception and Performance*, 7(3):573–579, 1981.

532. K. Hajek. Detection of logical coupling based on product release history. Thesis (m.s.), Technical University of Vienna, 1998.

533. H. W. Hake and W. R. Garner. The effect of presenting various numbers of discrete steps on scale reading accuracy. *Journal of Experimental Psychology*, 42(5):358–366, 1951.

534. G. S. Halford, W. H. Wilson, and S. Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral & Brain Sciences*, 21(6):803–831, 1998.

535. R. J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.

536. M. A. K. Halliday and R. Hasan. *Cohesion in English*. Pearson Education Limited, 1976.

537. D. Z. Hambrick and R. W. Engle. Effect of domain knowledge, working memory capacity, and age on cognitive performance: An investigation of the knowledge-is-power hypothesis. *Cognitive Psychology*, 44(4):339–387, 2002.

538. K. R. Hammond, R. M. Hamm, J. Grassia, and T. Pearson. Direct comparison of the efficacy of intuitive and analytical cognition in expert judgment. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17:753–770, Sept. 1987.

539. R. E. Hank. *Region-based compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

540. J. R. Hanley and P. Morris. The effects of amount of processing on recall and recognition. *Quarterly Journal of Experimental Psychology*, 39A:431–449, 1987.

541. C. L. Hardin and L. Maffi. *Color categories in thought and language*. Cambridge University Press, 1997.

542. T. Harley. *The Psychology of Language*. Psychology Press, second edition, 2001.

543. W. S. Harley. *Associative Memory in Mental Arithmetic*. PhD thesis, Johns Hopkins University, Oct. 1991.

544. M. W. Harm. *Division of Labor in a Computational Model of Visual Word Recognition*. PhD thesis, University of Southern California, Aug. 1998.

545. M. J. Harrold, J. A. Jones, and G. Rothermel. Empirical studies of control dependence graph size for C. *Empirical Software Engineering Journal*, 3(2):203–211, Mar. 1998.

546. J. Hartman. *Automatic control understanding for natural programs*. PhD thesis, University of Texas at Austin, May 1991.

547. A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. Technical Report Research Report 96/3, Compaq Western Research Laboratory, 1996.

548. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, pages 125–136, Jan. 1992.

549. L. Hatton. *Safer C : Developing Software for High-integrity and Safety-critical Systems*. McGraw–Hill, 1995.

550. L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, Mar. 1997.

551. L. Hatton. The T-experiments: Errors in scientific software. *IEE Computational Science & Engineering*, 4(2):27–38, Jan. 1997.

552. J. Hauser. Softfloat-2b. www.jhauser.us/arithmetic/SoftFloat.html, 2002.

553. J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, Mar. 1996.

554. J. A. Hawkins and G. Gilligan. Prefixing and suffixing universals in relation to basic word order. *Lingua*, 74:219–258, 1988.

555. B. Hayes. Third base. *American Scientist*, 89(6):490–494, 2001.

556. A. F. Healy and T. F. Cunningham. A developmental evaluation of the role of word shape in word recognition. *Memory & Cognition*, 20(2):141–150, 1992.

557. C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.

558. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.

559. L. Henderson and S. E. Henderson. Visual comparison of words and random letter strings: Effects of number and position of letters different. *Memory & Cognition*, 3(1):97–101, 1975.

560. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.

561. M. Henricson and E. Nyquist. *Industrial Strength C++, Rules and Recommendations*. Prentice Hall, Inc, 1997.

562. S. Henser. Thinking in Japanese? What have we learned about language-specific thought since Ervin Tripp's psychological tests of Japanese-English bilinguals. Technical Report No. 32, Nissan Institute of Japanese Studies, 2000.

563. R. N. A. Henson. *Short-term Memory for Serial Order*. PhD thesis, University of Cambridge, Nov. 1996.

564. R. N. A. Henson. Item repetition in short-term memory: Ranschburg repeated. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 24(5):1162–1181, 1998.

565. D. M. B. Herbert and J. S. Burt. What do students remember? Episodic memory and the development of schematization. *Applied Cognitive Psychology*, 18(1):77–88, 2004.

566. C. M. Herdman, D. Chernecki, and D. Norris. Naming cAsE aLtErNaTeD words. *Memory & Cognition*, 27(2):254–266, 1999.

567. C. M. Herdman and A. R. Dobbs. Attentional demands of visual word recognition. *Journal of Experimental Psychology: Human Perception and Performance*, 15(1):124–132, 1989.

568. D. S. Herrmann. *Software Safety and Reliability*. IEEE Computer Society, 1999.

569. R. Hertwig and G. Gigerenzer. The 'conjunction fallacy' revisited: How intelligent inferences look like reasoning errors. *Journal of Behavioral and Decision Making*, 12(2):275–305, 1999.

570. R. Hertwig and P. M. Todd. More is not always better: The benefits of cognitive limits. In L. Macchi and D. Hardman, editors, *The psychology of reasoning and decision making: A handbook*. John Wiley & Sons, Inc, 2000?

571. Hewlett-Packard. *PA-RISC 2.0*. Hewlett-Packard, 2.0 edition, 1995.

572. Y. Hida, X. S. Li, and D. H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, 2000.

573. T. P. Hill. A statistical derivation of the significant-digit law. *Statistical Science*, 10:354–363, 1996.

574. T. P. Hill. The first-digit phenomenon. *American Scientist*, 86:358–363, July-Aug. 1998.

575. D. J. Hilton. The social context of reasoning: Conversational inference and rational judgment. *Psychological Bulletin*, 118(2):248–271, 1995.

576. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61. ACM, June 2001.

577. M. Hind and A. Pioli. Which pointer analysis should I use? *International Symposium on Software Testing and Analysis (ISSTA 2000), Aug 2000*, 2000.

578. Hitachi Ltd. *H8S, H8/300 Series C Compiler User's Manual*. Hitachi Ltd, 4.0 edition, Sept. 1998.

579. M. H. Hodge and F. M. Pennington. Some studies of word abbreviation behavior. *Journal of Experimental Psychology*, 98(2):350–361, 1973.

580. D. D. Hoffman. *Visual Intelligence: How We Create What We See*. W. W. Norton, 2000.

581. R. M. Hogarth and H. J. Einhorn. Order effects in belief updating: The belief-adjustment model. *Cognitive Psychology*, 24:1–55, 1992.

582. R. M. Hogarth, C. R. M. McKenzie, B. J. Gibbs, and M. A. Marquis. Learning from feedback: Exactness and incentives. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(4):734–752, 1991.

583. M. B. Holbrook. A comparison of methods for measuring the interletter similarity between capital letters. *Perception & Psychophysics*, 17(6):532–536, 1975.

584. M. B. Holbrook. Effect of subjective interletter similarity, perceived word similarity, and contextual variables on the recognition of letter substitutions in a proofreading task. *Perceptual and Motor Skills*, 47:251–258, 1978.

585. J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. *Induction*. The MIT Press, 1989.

586. J. Hollis and T. Valentine. Proper name processing: Are proper names pure referencing expressions? *Journal of Experimental Psychology: Learning, Memory & Cognition*, 27:99–116, 2001.

587. A. Holm and B. Dodd. The effect of first written language on the acquisition of English literacy. *Cognition*, 59:119–147, 1996.

588. D. Holmes and M. C. McCabe. Improving precision and recall for soundex retrieval. In *Proceedings of the 2002 IEEE International Conference on Information Technology - Coding and Computing (ITCC)*, pages 22–27, Apr. 2002.

589. J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software–Practice and Experience*, 29(11):1005–1023, Sept. 1999.

590. A. A. Hook, B. Brykczynski, C. W. McDonald, S. H. Nash, and C. Youngblut. A survey of computer programming languages currently used in the department of defense. Technical Report P-3054, Institute for Defense Analyse, Jan. 1995.

591. R. Hoosain. Correlation between pronunciation speed and digit span size. *Perception and Motor Skills*, 55:1128–1128, 1982.

592. R. Hoosain and F. Salili. Language differences, working memory, and mathematical ability. In M. M. Grunberg, P. E. Morris, and R. N. Sykes, editors, *Practical aspects of memory: Current research and issues*, volume 2, pages 512–517. John Wiley & Sons, Inc, 1988.

593. J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 4^{th} Symposium on Software Testing, Analysis, and Verification*, pages 87–97. ACM Press, Oct. 1991.

594. L. M. Horowitz. Free recall and ordering of trigrams. *Journal of Experimental Psychology*, 62(1):51–57, 1961.

595. M. R. Horton. *Portable C Software*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

596. S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, Jan. 1997.

597. M. W. Howard and M. J. Kahana. Context variability and serial position effects in free recall. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 25(4):923–941, 1999.

598. M. W. Howard and M. J. Kahana. When does semantic similarity help episodic retrieval? *Journal of Memory and Language*, 46:85–98, 2002.

599. D. H. Howes and R. L. Solomon. Visual duration threshold as a function of word-probability. *Journal of Experimental Psychology*, 41:401–410, 1951.

600. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.

601. C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–8, 2003.

602. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. Technical Report DCS-TR-419, Department of Computer Science, Rutgers University, 2000.

603. C.-W. Hue and J. R. Erickson. Short-term memory for Chinese characters and radicals. *Memory & Cognition*, 16(3):196–205, 1988.

604. A. J. Hull. A letter-digit metric of auditory confusions. *British Journal of Psychology*, 64(4):579–585, 1973.

605. D. A. Hull and G. Grefenstette. A detailed analysis of English stemming algorithms. Technical Report MLTT-023, Xerox Research and Technology, Jan. 1996.

606. C. Hulme, S. Maughan, and G. D. A. Brown. Memory for familiar and unfamiliar words: Evidence for a long-term memory contribution to short-term memory span. *Journal of Memory and Language*, 30:685–701, 1991.

607. E. Hunt. The Whorfian hypothesis: A cognitive psychology perspective. *Psychological Review*, 98(3):377–389, 1991.

608. E. Hunt, C. Lunneborg, and J. Lewis. What does it mean to be high verbal? *Cognitive Psychology*, 7:194–227, 1975.

609. J. Huttenlocher. Constructing spatial images: A strategy in reasoning. *Psychological Review*, 75(6):550–560, 1968.

610. J. Hyönä. Do irregular letter combinations attract readers' attention? Evidence from fixation locations in words. *Journal of Experimental Psychology: Human Perception and Performance*, 21(1):68–81, 1995.

611. J. Hyönä, P. Niemi, and G. Underwood. Reading long words embedded in sentences: Informativeness of word halves affects eye movements. *Journal of Experimental Psychology: Human Perception and Performance*, 15(1):142–152, 1989.

612. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.

613. IBM. *Developing PowerPC Embedded Application Binary (EABI) Compliant Programs*. IBM, Sept. 1998.

614. IBM. *Preprocessing Directives - #pragma*. International Business Machines, 5.0 edition, 2000.

615. IBM. *z/Architecture: Principles of Operation*. International Business Machines, first edition, Dec. 2000.

616. IBM. *ILE C/C++ Compiler Reference*. IBM Canada Ltd, Ontario, Canada, sc09-48 16-00 edition, May 2001.

617. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.

618. IBM Canada Ltd. *C for AIX Compiler Reference*. International Business Machines Corporation, May 2002.

619. J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991.

620. IEEE. *IEEE 1003.3 Standard for Information Technology —Test Methods for Measuring Conformance to POSIX.1*. IEEE, 1991.

621. Imsys AB. *the Cjip Technical Reference Manual*. Imsys AB, Sweden, v0.23 edition, 2001.

622. INMOS Limited. *Transputer Reference Manual*. Prentice–Hall, 1988.

623. T. Instruments. *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*. Texas Instruments, Inc, spru024e edition, Aug. 1999.

624. Intel. *i860 64-bit microprocessor programmer's reference manual*. Intel, Inc, 1989.

625. Intel. *MCS 51 Microcontroller Family User's Manual*. Intel, Inc, 272383-002 edition, Feb. 1994.

626. Intel. *Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*. Intel Corporation, 1.1 edition, Jan. 1999.

627. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.

628. Intel. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Intel, Inc, Feb. 2001.

629. Intel. *Intel XScale Microarchitecture Programmers Reference Manual*. Intel, Inc, 2001.

630. Intel. *Intel C++ Compiler User's Guide*. Intel, Inc, 2002.

631. Intel, Inc. *Intel IA-64 Architecture Software Developer's Manual*, 2000. Instruction Set Reference.

632. K. Ishihara, T. Okada, and S. Matsui. English vocabulary recognition and production: A preliminary survey report. *Doshisha Studies in Language and Culture*, 2(1):143–175, 1999.

633. ISO. *ISO 6160-1979(E) —Programming languages —PL/1*. ISO, 1979.

634. ISO. *ISO 1989-1985(E) —Programming languages —COBOL*. ISO, 1985.

635. ISO. *Implementation of ISO/IEC TR 10034:1990 Guidelines for the preparation of conformity clauses in programming language standards*. ISO, 1990.

636. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.

637. ISO. *ISO/IEC Guide 25:1990 General requirements for the competence of calibration and testing laboratories*. ISO, 1990.

638. ISO. *Implementation of ISO/TR 9547:1988 Programming language processors —Test methods —Guidelines for their development and acceptability*. ISO, 1991.

639. ISO. *ISO/IEC 10206:1991 Information technology —Programming languages —Extended Pascal*. ISO, 1991.

640. ISO. *ISO/IEC 1539:1992 Information technology —Programming languages —FORTRAN*. ISO, 1991.

641. ISO. *ISO/IEC 9075:1992(E) Information technology —Database languages —SQL*. ISO, 1992.

642. ISO. *ISO/IEC 10967-1:1994(E) Information technology —Language independent arithmetic —Part 1: Integer and floating point arithmetic*. ISO, 1994.

643. ISO. *Implementation of ISO/IEC 8651-4:1995 Information technology —Computer graphics —Graphics kernel system (GKS) language bindings —Part 4: C*. ISO, 1995.

644. ISO. *ISO/IEC 13211-1:1995 Information technology —Programming languages, their environments and system software interfaces —Programming language Prolog —Part 1: General Core*. ISO, 1995.

645. ISO. *ISO/IEC 8652:1995(E) Information technology —Programming languages —Annotated Ada Reference Manual*. ISO, 1995.

646. ISO. *ISO/IEC 10514-1:1996 Information technology —Programming languages —Part 1. Modula-2, Base language*. ISO, 1996.

647. ISO. *ISO/IEC 13817-1:1996 Information technology —Programming languages, their environments and system software interfaces —Vienna Development Method —Specification language Part 1. Base language*. ISO, 1996.

648. ISO. *ISO/IEC 14977:1996 Information technology —Syntactic metalanguage —Extended BNF*. ISO, 1996.

649. ISO. *Implementation of ISO/IEC TR 10176:1997 Information technology —Guidelines for the preparation of programming language standards*. ISO, 1997.

650. ISO. *ISO/IEC 13816:1997 Information technology —Programming languages, their environments and system software interfaces —Programming language ISLISP*. ISO, 1997.

651. ISO. *ISO TR 15580:1998 Information technology —Programming languages —Fortran —Floating-point exception handling*. ISO, 1998.

652. ISO. *ISO/IEC 1539-3:1998 Information technology —Programming languages —FORTRAN —Part 3: Conditional compilation*. ISO, 1998.

653. ISO. *ISO/IEC 13210:1999 Information technology —Requirements and guidelines for test methods specifications and test method implementation for measuring conformance to POSIX standards*. ISO, 1999.

654. ISO. *ISO/IEC 13751.2:2000 Information technology —Programming languages, their environments and system software interfaces —Programming language Extended APL*. ISO, 2000.

655. ISO. *ISO/IEC TR 15942:2000 Programming languages —Guide for the Use of the Ada Programming Language in High Integrity Systems*. ISO, 2000.

656. ISO. *ISO/IEC 9496:2003 CHILL —The ITU-T programming language*. ISO, 2003.

657. ISO. *Programming languages, their environments and system software interfaces —Extensions for the programming language C to support embedded processors*. ISO, 2004.

658. S. A. J. The seer-sucker theory: The value of experts in forecasting. *Technology Review*, pages 16–24, June-July 1980.

659. W. P. J and M. A. McDaniel. Pictorial enhancement of text memory: Limitations imposed by picture type and comprehension skill. *Memory & Cognition*, 20(5):472–482, 1992.

660. A. M. Jacobs and J. Grainger. Models of visual word recognition: Sampling the state of the art. *Journal of Experimental Psychology: Human Perception and Performance*, 20(6):1311–1334, 1994.

661. R. Jaeschke. *Portability and the C Language*. Hayden Books, 4300 West 62nd Street, Indianapolis, IN 46268, USA, 1989.

662. P. J. Jalics. COBOL on a PC: A new perspective on a language and its performance. *Communications of the ACM*, 30(2):142–154, Feb. 1987.

663. C. T. James. Vowels and consonants as targets in the search of single words. *Bulletin of the Psychonomic Society*, 2(4B):402–404, 1974.

664. C. T. James and D. E. Smith. Sequential dependencies in letter search. *Journal of Experimental Psychology*, 85(1):56–60, 1970.

665. Java Grande Forum Numerics Working Group. Improving java for numerical computation. `math.nist.gov/javanumerics/reports/jgfnwg-01.htm`, Oct. 1998.

666. J. J. Jenkins. Remember that old theory of memory? Well, forget it! *American Psychologist*, 29(11):785–795, 1974.

667. L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, pages 55–64, Sept. 2007.

668. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

669. L. Jin, C. Li, and S. Mehrota. Efficient similarity string joins in large data sets. Technical Report TR-DB-02-04, University of California at Irvine, Feb. 2002.

670. S. Jinturkar. *Data-Specific Optimizations*. PhD thesis, University of Virginia, May 1996.

671. M. K. Johansen and T. J. Palmeri. Are there representational shifts during category learning? *Cognitive Psychology*, 45(4):482–553, 2002.

672. S. C. Johnson. Lint, a C program checker. Technical Report Computing Science Technical Report No.65, Bell Telephone Laboratories, 1977.

673. S. C. Johnson. A tour through the portable C compiler. In B. W. Kernighan and M. D. McIlroy, editors, *Unix Programmer's Manual, 7th edition, Volume 2B*, chapter 33. Bell Laboratories, Murray Hill, NJ, Jan. 1979. Republished by Holt, Rinehart and Winston, New York, ISBN 0-03-061743-X, 1983.

674. S. C. Johnson and B. W. Kernighan. The programming language B. Technical Report 8, Bell Telephone Laboratories, Jan. 1973.

675. S. C. Johnson and D. M. Ritchie. The C language calling sequence. Technical Report Computing Science Technical Report No. 102, Bell Laboratories, Sept. 1981.

676. W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc, 1986.

677. P. N. Johnson-Laird. *Mental Models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983.

678. P. N. Johnson-Laird. Mental models and deduction. *TRENDS in Cognitive Science*, 5(10):434–442, 2001.

679. M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the first International Symposium on Memory management*, pages 26–36, Mar. 1998.

680. C. Jones. *Programming Productivity*. McGraw–Hill Book Company, 1986.

681. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.

682. D. M. Jones. Who guards the guardians? www.knosof.co.uk/whoguard.html, 1992.

683. D. M. Jones. The open systems portability checker reference manual. www.knosof.co.uk, 1999.

684. D. M. Jones. The 7±2 urban legend. MISRA C 2002 conference http://www.knosof.co.uk/cbook/misart.pdf, Oct. 2002.

685. D. M. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, Aug. 2006.

686. D. M. Jones. Operand names influence operator precedence decisions. *C Vu*, 20(1):??–??, Feb. 2008.

687. D. W. Jones. BCD arithmetic, a tutorial. www.cs.uiowa.edu/~jones, July 1999.

688. J. T. Jones, B. W. Pelham, M. C. Mirenberg, and J. J. Hetts. Name letter preferences are not merely mere exposure: Implicit egotism as self-regulation. *Journal of Experimental Psychology*, 38:170–177, 2002.

689. R. Jones and R. Lims. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Addison–Wesley, 1996.

690. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.

691. J. Jonides and C. M. Jones. Direct coding for frequency of occurrence. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 18(2):368–378, 1992.

692. R. Jordan, R. Lotufo, and D. Argiro. *Khoros Pro 2001 Student Version*. Khoral Research, Inc, 2000.

693. M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.

694. M. Jørgensen and D. I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance: Research and Practice*, 14(2):123–146, 2002.

695. N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *Proceedings of Workshop of Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

696. M. A. Just and P. A. Carpenter. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior*, 10:244–253, 1971.

697. M. A. Just and P. A. Carpenter. A capacity theory of comprehension: Individual differences in working memory. *Psychological Review*, 99(1):122–149, 1992.

698. F. Justicia, S. Defior, S. Pelegrina, and F. J. Martos. The sources of error in Spanish writing. *Journal of Research in Reading*, 22(2):198–202, 1999.

699. D. M. Kagan and J. M. Douthat. Personality and learning FORTRAN. *International Journal of Man-Machine Studies*, 22(4):395–402, 1985.

700. W. Kahan. IEEE standard 754 for binary floating-point arithmetic. Lecture Notes in progress, May 1995.

701. W. Kahan. How Java's floating-point hurts everyone. In *ACM 1998 Workshop on Java for High-Performance Computing*, 1998.

702. W. Kahan. The improbability of PROBABILISTIC ERROR ANALYSIS for numerical computation. University of California at Berkeley, 1998.

703. W. Kahan. Miscalculating area and angles of a needle-like triangle. Lecture Notes for Introductory Numerical Analysis Classes, Mar. 2000.

704. M. J. Kahana. Associative symmetry and memory theory. *Memory & Cognition*, 30(6):823–840, 2002.

705. D. Kahneman, P. Slovic, and A. Tversky, editors. *Judgment under uncertainty: Heuristics and biases*. Cambridge University Press, 1982.

706. D. Kahneman and A. Tversky. On the psychology of prediction. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 4, pages 48–68. Cambridge University Press, 1982.

707. D. Kahneman and A. Tversky. Subjective probability: A judgment of representativeness. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 3, pages 32–47. Cambridge University Press, 1982.

708. D. Kahneman and A. Tversky. Choices, values, and frames. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 1, pages 1–16. Cambridge University Press, 1999.

709. D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 2, pages 17–43. Cambridge University Press, 1999.

710. S. Kahrs. Mistakes and ambiguities in the definition of standard ML. Technical Report LFCS report ECS-LFCS-93-257, University of Edinburgh, Scotland, Apr. 1993.

711. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

712. Y. Kareev. Seven (indeed, plus or minus two) and the detection of correlations. *Psychological Review*, 107(2):397–402, 2000.

713. M. B. Karlin and G. H. Bower. Semantic category effects in visual word search. *Perception & Psychophysics*, 19(5):417–424, 1976.

714. M. Karnaugh. The map method for synthesis of combinational logic circuits. *AIEE Transactions Comm. Elec*, 72:593–599, 1953.

715. M. Kawakami. Effects of phonographic and phonological neighbors on katakana word recognition. In *The Second International Conference on Cognitive Science and The* 16^{th} *Annual Meeting of the Japanese Cognitive Science Society Joint Conference (ICCS/JCSS99)*. Japanese Cognitive Science Society, July 1999.

716. R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

717. Keil. *C Compiler manual*. Keil Software, Inc, ??? edition, May 2005.

718. B. Kelk. Letter frequency rankings for various languages. www.bckelk.uklinux.net/words/etaoin.html, 2003.

719. R. Kelsey, W. Clinger, J. Rees, H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele JR., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. Technical report, Feb. 1998.

720. C. F. Kemerer and S. Slaughter. Determinants of software maintenance profiles: An empirical investigation. *Software Maintenance: Research and Practice*, 9(4):235–251, 1997.

721. C. K. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–503, 1999.

722. R. L. Kennell and R. Eigenmann. Automatic parallelization of C by means of language transcription. In S. Chatterjee, J. Prins, L. Carter, J. Ferrante, Z. Li, D. C. Sehr, and P.-C. Yew, editors, *Proceedings of the* 11^{th} *International Workshop on Languages and Compilers for Parallel Computing (LCPC-98)*, Lecture Notes in Computer Science, pages 166–180. Springer, 1998.

723. B. W. Kernighan. Programming in C-A tutorial. Technical Report ???, Bell Laboratories, Aug. ???

724. B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison–Wesley, 1999.

725. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc, 1978.

726. B. W. Kernighan and D. M. Ritchie. Recent changes to C. from Ritchie's web page, Nov. 1978.

727. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc, 1988.

728. M. D. Kernighan, K. W. Church, and W. A. Gale. A spelling correction program based on a noisy channel model. In *Proceedings of COLING-90*, pages 205–210, 1990.

729. B. Kessler. Phonetic comparison algorithms. *Transactions of the Philological Society*, 103(2):243–260, 2005.

730. B. Kessler and R. Treiman. Syllable structure and phoneme distribution. *Journal of Memory and Language*, 37:295–311, 1997.

731. D. E. Kieras and D. E. Meyer. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. Technical Report TR-95/ONR-EPIC-5, University of Michigan, 1995.

732. J. King and M. A. Just. Individual differences in syntactic processing: The role of working memory. *Journal of Memory and Language*, 30:580–602, 1991.

733. G. C. Kinney and D. J. Showman. The relative legibility of upper case and lower case typewritten words. *Information Display*, 4:34–39, 1967.

734. W. Kintsch. *Comprehension: A paradigm for cognition*. Cambridge University Press, 1998.

735. W. Kintsch and J. Keenan. Reading rate and retention as a function of the number of propositions in the base structure of sentences. *Cognitive Psychology*, 5:257–274, 1973.

736. W. Kintsch, E. Kozminsky, W. J. Streby, G. McKoon, and J. M. Keenan. Comprehension and recall of text as a function of content variables. *Journal of Verbal Learning and Verbal Behavior*, 14:196–214, 1975.

737. W. Kintsch, T. S. Mandel, and E. Kozminsky. Summarizing scrambled stories. *Memory & Cognition*, 5(5):547–552, 1977.

738. S. Kirby. Language evolution without natural selection: From vocabulary to syntax in a population of learners. Technical Report Edinburgh Occasional Papers in Linguistics, Edinburgh University, Apr. 1998.

739. D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 204–213. Association for Computing Machinery, 1997.

740. G. Kiss, C. Armstrong, R. Milroy, and J. Piper. An associative thesaurus of English and its computer analysis. In A. J. Aitken, R. W. Bailey, and N. Hamilton-Smith, editors, *The Computer and Literary Studies*, pages 153–165. Edinburgh University Press, 1973.

741. T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, 2000.

742. T. Kistler and M. Franz. Continuous program optimization: A case study. Technical Report Technical Report No. 00-19, Department of Information and Computer Science, University of California, Irvine, 2000.

743. S. Kitayama and M. Karasawa. Implicit self-esteem in Japan: Name-letters and birthday numbers. *Personality & Social Psychology Bulletin*, 23(7):736–742, 1997.

744. D. Klahr, W. G. Chase, and E. A. Lovelace. Structure and process in alphabetic retrieval. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(3):462–477, 1983.

745. J. Klayman and Y.-W. Ha. Confirmation, disconfirmation, and information in hypothesis testing. *Psychological Review*, 94(2):211–228, 1987.

746. J. Klayman, J. B. Soll, C. Gonz/'alez-Vallejo, and S. Barlas. Overconfidence: It depends on how, what, and whom you ask. *Organizational Behavior and Human Decision Processes*, 79(3):216–247, 1999.

747. G. Klein. *Sources of Power*. The MIT Press, 1999.

748. M. Knauff, R. Rauh, and C. Schlieder. Prefered mental models in qualitative spatial reasoning: A cognitive assessment of Allen's calculus. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 200–205. Lawrence Erlbaum Associates, 1995.

749. J. L. Knetsch. The endowment effect and evidence of nonreversible indifference curves. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 9, pages 171–179. Cambridge University Press, 1999.

750. D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.

751. D. E. Knuth. Structure programming with `go to` statements. *Computing Surveys*, 6(4):261–301, Dec. 1974.

752. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

753. D. E. Knuth. The errors of TEX. *Software–Practice and Experience*, 19(7):607–685, 1989.

754. K. Koda. Effects of L1 orthographic representation on L2 phonological coding strategies. *Journal of Psycholinguistic Research*, 18(2):201–220, 1989.

755. B. Koehler and R. N. Horspool. CCC: A caching compiler for C. *Software–Practice and Experience*, 27(2):155–165, 1997.

756. J. J. Koehler. The base rate fallacy reconsidered: Descriptive, normative and methodological challenges. *Behavior & Brain Sciences*, 19(1):1–17, 1996.

757. A. Koenig. *C Traps and Pitfalls*. Addison–Wesley, 1989.

758. D. Koes, M. Budiu, G. Venkataramani, and S. C. Goldstein. Programmer specified pointer independence. Technical Report CMU-CS-03-128, Carnegie Mellon University, Apr. 2003.

759. C. Kohsom and F. Gobet. Adding spaces to Thai and English: Effects on reading. In *Proceedings of the* 19^{th} *Annual Meeting of Cognitive Science Society*, 1997.

760. P. A. Kolers. Reading A year later. *Journal of Experimental Psychology: Human Learning and Memory*, 2(3):554–565, 1976.

761. P. A. Kolers and D. N. Perkins. Spatial and ordinal components of form perception and literacy. *Cognitive Psychology*, 7:228–267, 1975.

762. P. J. Koopman Jr. *Stack Computers the new wave*. Mountain View Press, 1989.

763. M. Koppel, J. Schler, and K. Zigdon. Determining an author's native language by mining a text for errors. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 624–628, Aug. 2005.

764. A. Koriat. Phonetic symbolism and feeling of knowing. *Memory & Cognition*, 3(5):545–548, 1975.

765. A. Koriat. How do we know that we know? The accessibility model of the feeling of knowing. *Psychological Review*, 100(4):609–639, 1993.

766. A. Koriat, M. Goldsmith, and A. Pansky. Toward a psychology of memory accuracy. *Annual Review of Psychology*, 51:481–537, 2000.

767. S. M. Kosslyn and S. P. Shwartz. Empirical constraints on theories of visual imagery. In J. Long and A. D. Baddeley, editors, *Attention and Performance IX*, pages 241–260. Lawrence Erlbaum Associates, 1981.

768. R. J. Koubek, W. K. LeBold, and G. Salvendy. Predicting performance in computer programming courses. *Behavior and Information Technology*, 4(2):113–129, 1985.

769. H. Kozima and T. Furugori. Similarity between words computed by spreading activation on an English dictionary. In *Proceedings of the* 6^{th} *Conference of the European Chapter of the Association for Computational Linguistics (EACL-93)*, pages 232–239, 1993.

770. H. Kozima and A. Ito. Context-sensitive measurement of word distance by adaptive scaling of a semantic space. In R. Mitkov and N. Nicolov, editors, *Recent Advances in Natural Languages processing: Selected Papers from RANLP'95*, chapter 2, pages 111–124. John Benjamins Publishing Company, 1996.

771. D. S. Kreiner and P. B. Gough. Two ideas about spelling: Rules and word-specific memory. *Journal of Memory and Language*, 29:103–118, 1990.

772. C. B. Kreitzberg and B. Shneiderman. *The elements of FORTRAN style: techniques for effective programming*. Harcourt, Brace, Jovanovich, San Diego, CA, USA, 1972.

773. J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 301–309, Oct. 2001.

774. R. Krovetz. Viewing morphology as an inference process. Technical Report UM-CS-1993-036, University of Mass-Amherst, Apr. 1993.

775. I. Krsul. Authorship analysis: Identifying the author of a program. Technical Report Purdue Technical Report CSD-TR-94-030, Department of Computer Sciences, Purdue University, 1994.

776. I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. Technical Report Technical Report TR-96-052, Department of Computer Sciences, Purdue University, 1996.

777. L. E. Krueger. A theory of perceptual matching. *Psychological Review*, 85(4):278–304, 1978.

778. N. kuan Tsao. On the distribution of significant digits and round-off errors. *Communications of the ACM*, 17(5):269–271, May 1974.

779. M. Kubovy and S. Gepshtein. Gestalt: From phenomena to laws. In K. L. Boyer and S. Sarkar, editors, *Perceptual Organization for Artificial Vision Systems*, chapter 5, pages 41–71. Kluwer Academic Publishers, Boston, 2000.

780. H. Kucera and W. N. Francis. *Computational analysis of present-day American English*. Brown University Press, 1967.

781. D. J. Kuck, D. S. Parker Jr., and A. H. Sameh. Analysis of rounding methods in floating-point arithmetic. *IEEE Transactions on Computers*, C-26(7):643–650, July 1977.

782. T. Kuennapas and A.-J. Janson. Multidimensional similarity of letters. *Perceptual and Motor Skills*, 28:3–12, 1969.

783. K. Kukich. Spelling correction for the telecommunications network for the deaf. *Communications of the ACM*, 35(5):80–99, May 1992.

784. K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

785. K.-I. Kum, J. Kang, and W. Sung. A floating-point to fixed-point C converter for fixed-point digital signal processors. In *Proceedings of the Second SUIF Compiler Workshop*, Aug. 1997.

786. K. Kunchithapadam and J. R. Larus. Using lightweight procedures to improve instruction cache performance. Technical Report CS-Technical-Report 1390, University of Wisconsin-Madison, Jan. 1999.

787. P. Kundu and B. B. Chaudhuri. Error pattern in Bangla text. *International Journal of Dravidian Linguistics*, 28(2):49–88, 1999.

788. S. M. Kurlander, T. A. Proebsting, and C. N. Fischer. Efficient instruction scheduling for delayed-load architectures. *ACM Transactions on Programming Languages and Systems*, 17(5):740–776, 1995.

789. W. Labov. The boundaries of words and their meaning. In C.-J. N. Bailey and R. W. Shuy, editors, *New ways of analyzing variation of English*, pages 340–373. Georgetown Press, 1973.

790. B. Laguë, C. Ledue, A. L. Bob, E. Merlo, and M. Dagenais. An analysis framework for understanding layered software architectures. In 6^{th} *International Workshop on Program Comprehension*, 1998.

791. B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings: 1997 International Conference on Software Maintenance*, pages 314–321. IEEE Computer Society Press, 1997.

792. O. Laitenberger and J.-M. DeBaud. Perspective-based reading of code documents at Robert Bosch GmbH. Technical Report Technical Report ISERN-97-14, Fraunhofer Institute for Experimental Software Engineering, 1997.

793. O. Laitenberger and J.-M. DeBaud. An encompassing life-cycle centric survey of software inspection. Technical Report Technical Report ISERN-98-32, Fraunhofer Institute for Experimental Software Engineering, 1998.

794. O. Laitenberger, K. E. Emam, and T. Harbich. An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents. Technical Report Technical Report ISERN-99-01, Fraunhofer Institute for Experimental Software Engineering, 1999.

795. K. Laitinen. *Natural naming in software development and maintenance*. PhD thesis, University of Oulu, Finland, Oct. 1995. VTT Publications 243.

796. K. Laitinen, J. Taramaa, M. Heikkilä, and N. C. Rowe. Enhancing maintainability of source programs through disabbreviation. *Journal of Systems and Software*, 37:117–128, 1997.

797. G. Lakoff and M. Johnson. *Metaphors We Live By*. The University of Chicago Press, 1980.

798. J. Lakos. *Large Scale C++ Software Design*. Addison–Wesley, 1996.

799. B. L. Lambert, K.-Y. Chang, and P. Gupta. Effects of frequency and similarity neighborhoods on phamacists' visual perception of drug names. *Social Science and Medicine*, 57(10):1939–1955, Nov. 2003.

800. B. L. Lambert, D. Donderi, and J. W. Senders. Similarity of drug names: Comparison of objective and subjective measures. *Psychology and Marketing*, 19(7-8):641–662, 2002.

801. T. K. Landauer. How much do people remember? Some estimates of the quantity of learned information in long-term memory. *Cognitive Science*, 10:477–493, 1986.

802. T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

803. T. K. Landauer and L. A. Streeter. Structural differences between common and rare words: Failure of equivalence assumptions for theories of word recognition. *Journal of Verbal Learning and Verbal Behavior*, 12:119–131, 1973.

804. W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

805. W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the* 18^{th} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, 1991.

806. D. Landy and R. L. Goldstone. The alignment or ordering and space in arithmetic computation. In *Proceedings of the Twenty-Ninth Annual Meeting of the Cognitive Science Society*, pages 437–442, Aug. 2007.

807. G. Langdale. *The Effect of Profile Choice and Profile Gathering Methods on Profile-Driven Optimization Systems*. PhD thesis, Carnegie Mellon University, Oct. 2003.

808. M. Lange and A. Content. The grapho-phonological system of written French: Statistical analysis and empirical evaluation. In *ACL'99: The* 37^{th} *Annual Meeting of the Association for Computational Linguistics*, pages 436–442, June 1999.

809. E. J. Langer. The illusion of control. *Journal of Personality and Social Psychology*, 32(2):311–328, 1975.

810. D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable code generation for embedded DSP processors. In P. Merwedel and G. Goossens, editors, *Code generation for embedded processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, July 1995.

811. J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

812. S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, Nov. 2001.

813. K. A. Latorella. Investigating interruptions: Implications for flightdeck performance. Technical Report NASA/TM-1999-209707, NASA, Oct. 1999.

814. F. Lavigne, F. Vitu, and G. d'Ydewalle. The influence of semantic context on initial eye landing sites in words. *Acta Psychologica*, 104:191–214, 2000.

815. O. Lawlor, H. Govind, I. Dooley, M. Breitenfeld, and L. Kale. Performance degradation in the presence of subnormal floating-point values. In *Proceedings of the International Workshop on Operating System Interference in High Performance Applications*, page ???, Sept. 2005.

816. A. R. Lebeck. Cache conscious programming in undergraduate computer science. In D. Joyce, editor, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 31.1 of *SIGCSE Bulletin*, pages 247–251, N. Y., Mar. 24–28 1999. ACM Press.

817. C. Lebiere. *The Dynamics of Cognition: An ACT-R Model of Cognitive Arithmetic*. PhD thesis, Carnegie Mellon University, Nov. 1998.

818. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 330–335. IEEE, Dec. 1997.

819. D. Lee, J.-L. Baer, B. Bershad, and T. Anderson. Reducing startup latency in web and desktop applications. Technical Report TR-99-03-01, University of Washington, Department of Computer Science and Engineering, Mar. 1999.

820. D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the* 25^{th} *Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 27–38, New York, June 27–July 1 1998. ACM Press.

821. D. J. Lee. Interpretation of morpheme rank ordering in L_2 research. In P. S. Dale and D. Ingram, editors, *Child Language – An International Perspective*, pages 261–271. Baltimore: University Park Press, 1981.

822. G. Leech, R. Garside, and M. Bryant. CLAWS4: The tagging of the British national corpus. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING 94)*, pages 622–628, Apr. 1994.

823. G. Leech, P. Rayson, and A. Wilson. *Word Frequencies in Written and Spoken English*. Pearson Education, 2001.

824. C. Leen, J. K. Lee, T. T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *Proceedings of the* 13^{th} *International Symposium on System Synthesis (ISSS'00)*, 2000.

825. J.-A. LeFevre, J. Bisanz, K. E. Daley, L. Buffone, S. L. Greenham, and G. S. Sadesky. Multiple routes to solution of single-digit

multiplication problems. *Journal of Experimental Psychology: General*, 125(3):284–306, 1996.

826. J.-A. LeFevre and J. Morris. More on the relation between division and multiplication in simple arithmetic: Evidence for mediation of division solutions via multiplication. *Memory & Cognition*, 27(5):803–812, 1999.

827. V. Lefèvre and J.-M. Muller. Worst case for correct rounding of the elementary functions in double precision. Technical Report No. 4004, Unité de recherche INRIA Rhône-Alpes, Nov. 2000.

828. C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 218–227, Toulouse, France, Jan. 8–12, 2000. IEEE Computer Society TCCA.

829. C. R. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.

830. G. E. Legge, T. S. Klitz, and B. S. Tjan. Mr. Chips: An ideal-observer model of reading. *Psychological Review*, 104(3):524–553, 1997.

831. D. R. Lehman, R. O. Lempert, and R. E. Nisbett. The effects of graduate training on reasoning. *American Psychologist*, 43(6):431–442, 1988.

832. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Wurski. Metrics and laws of software evolution - the nineties view. In 4^{th} *International Software Metrics Symposium (METRICS '97)*, pages 20–32. IEEE, Nov. 1997.

833. P. Lemaire, H. Abdi, and M. Fayol. The role of working memory resources in simple cognitive arithmetic. *European Journal of Cognitive Psychology*, 8(1):73–103, 1996.

834. P. Lemaire and M. Fayol. When plausibility judgments supersede fact retrieval: The example of the odd-even effect on product verification. *Memory & Cognition*, 23(1):34–48, 1995.

835. K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50:1174–1190, 2001.

836. X. Leroy. Formal certification of a compiler back-end *or*: Programming a compiler with a proof assistant. In *Proceedings of the 2006 POPL Conference*, pages 42–54, Jan. 2006.

837. T. C. Lethbridge. What knowledge is important to a software professional? *IEEE Computer*, 33(5):44–50, May 2000.

838. T. C. Lethbridge, S. E. Sim, and J. Singer. Software anthropology: Performing field studies in software companies. 2000.

839. S. Letovsky. Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Corporation, 1986.

840. S. Letovsky. Cognitive processes in program comprehension. *The Journal of Systems and Software*, 7(4):325–339, Dec. 1987.

841. R. Leupers. Exploiting conditional instructions in code generation for embedded VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999*, 1999.

842. R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings on* 11^{th} *international symposium on System synthesis*, pages 3–8, Dec. 1998.

843. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, Jan. 1998.

844. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, Jan. 1998.

845. C. Lever and D. Boreham. malloc() Performance in a multithreaded linux environment. Technical Report CITI Technical Report 00-5, University of Michigan, May 2000.

846. B. Leverett and T. G. Szymanski. Chaining span-dependent jump instructions. *ACM Transactions on Programming Languages and Systems*, 2(3):274–289, 1980.

847. B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. *Computer*, 13(8):38–49, 1980.

848. J. R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

849. P. Lewicki, T. Hill, and E. Bizot. Acquisition of procedural knowledge about a pattern stimuli that cannot be articulated. *Cognitive Psychology*, 20:24–37, 1988.

850. B. Li and R. Gupta. Bit section instruction set extension of ARM for embedded applications. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES 2002*, pages 69–78. ACM, Oct. 2002.

851. E. Y. Li, H.-G. Chen, and W. Cheung. Total quality management in software development process. *The Journal of the Quality Assurance Institute*, 14(1):4–6 & 35–41, Jan. 2000.

852. K.-C. Li and H. Schwetman. Implementing a scalar C compiler on the Cyber 205. *Software–Practice and Experience*, 14(9):867–888, 1984.

853. W. Li. Random texts exhibit Zipf's–law–like word frequency distribution. *IEEE Transactions on Information Theory*, 38(6):1842–1845, 1992.

854. Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Proceedings of the* 1^{st} *Workshop on Interaction between Compilers and Computer Architectures*, Feb. 1996.

855. S. Lichtenstein and B. Fishhoff. Do those who know more also know more about how much they know? *Organizational Behavior and Human Performance*, 20:159–183, 1977.

856. D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of Coling/ACL-98*, pages 768–774, 1998.

857. W.-Y. Lin. An optimizing compiler for the TMS320C25 DSP processor. Thesis (m.s.), University of Toronto, Department of Electrical and Computer Engineering, 1995.

858. M. A. Linton and R. W. Quong. A macroscopic profile of program compilation and linking. *IEEE Transactions on Software Engineering*, 15(4):427–436, Apr. 1989.

859. M. H. Lipasti. *Value locality and speculative execution*. PhD thesis, Carnegie Mellon University, Apr. 1997.

860. A. Litman. An implementation of precompiled headers. *Software–Practice and Experience*, 23(3):341–350, 1993.

861. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corporation, 1986.

862. A. F. Llitjós. Improving pronunciation accuracy of proper names with language origin classes. Thesis (m.s.), Carnegi Mellon University, PA, USA, Aug. 2001.

863. G. D. Logan and S. T. Klapp. Automatizing alphabet arithmetic: I. Is extended practice necessary or produce automaticity? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(2):179–195, 1991.

864. G. P. Logan. Toward an instance theory of automatization. *Psychological Review*, 95(4):492–527, 1988.

865. R. H. Logie, S. D. Sala, V. Wynn, and A. D. Baddeley. Visual similarity effects in immediate verbal serial recall. *The Quarterly Journal of Experimental Psychology*, 53A(3):626–646, 2000.

866. A. Loginov, S. H. Yong, S. Horowitz, and T. Reps. Debugging via run-time type checking. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering 4th International Conference (FASE 2001)*, pages 217–232. Springer-Verlag, Apr. 2001.

867. Longman. *Longman Dictionary of Contemporary English*. Longman, 2001.

868. D. P. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

869. E. A. Lovelace and S. S. Southall. Memory for words in prose and their locations on the page. *Memory & Cognition*, 11(5):429–434, 1983.

870. J. Lu. *Interprocedural Pointer Analysis for C*. PhD thesis, Rice University, Apr. 1998.

871. J. A. Lucy. *Language diversity and thought: A reformulation of the linguistic relativity hypothesis*. Cambridge University Press, 1992.

872. C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

873. G. Lukatela and M. T. Turvey. Reading in two alphabets. *American Psychologist*, 53(9):1057–1072, 1998.

874. P. Lukowicz, E. A. Heinz, L. Prechelt, and W. F. Tichy. Experimental evaluation in computer science: a quantitative study. Technical Report iratr-1994-17, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Feb. 1994.

875. K. Lunde. *Understanding Japanese Information Processing*. O'Reilly & Associates, Inc., 1993.

876. K. Lunde. *CJKV Information Processing*. O'Reilly & Associates, Inc., 1999.

877. G. Luo, T. Chen, and H. Yu. Toward a progress indicator for program compilation. *Software–Practice and Experience*, 37(???):909–933, Apr. 2007.

878. A. R. Luria. *The mind of a mnemonist*. Harvard University Press, 1986.

879. R. Lutz and S. Greene. Measuring phonological similarity: The case of personal names. Technical report, Language Analysis Systems, Inc., 2001.

880. J. N. MacGregor. Short-term memory capacity: Limitation or optimization? *Psychological Review*, 94(1):107–108, 1987.

881. L. MacKellar. Variations in productivity over the life span: A review and some implications. Technical Report IR-02-061, International Institute for Applied Systems Analysis, Austria, Sept. 2002.

882. I. S. MacKenzie and R. W. Soukoreff. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction*, 17:147–198, 2002.

883. C. M. MacLeod, E. B. Hunt, and N. N. Matthews. Individual differences in the verification of sentence-picture relationships. *Journal of Verbal Learning and Verbal Behavior*, 17:493–507, 1978.

884. W. T. Maddox and C. J. Bohil. Costs and benefits in perceptual categorization. *Memory & Cognition*, 28:597–615, 2000.

885. D. J. Magenheimer, L. Peters, K. W. Pettis, and D. Zuras. Integer multiplication and division on the HP precision architecture. *IEEE Transactions on Computers*, 37(8):980–990, 1988.

886. M. Magnus. *What's in a Word? Studies in Phonosemantics*. PhD thesis, Norwegian University of Department of Linguistics Science and Technology, Apr. 2001.

887. E. A. Maguire, D. G. Gadian, I. S. Johnsrude, C. D. Good, J. Ashburner, R. S. J. Frackowiak, and C. D. Frith. Navigation-related structural change in the hippocampi of taxi drivers. *Proceedings of the National Academy of Sciences*, 97(8):4398–4403, 2000.

888. M. A. Malcolm. On accurate floating-point summation. *Communications of the ACM*, 14(11):731–736, 1971.

889. M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM*, 15(11):949–951, 1972.

890. K. J. Malmberg, M. Steyvers, J. D. Stephens, and R. M. Shiffrin. Feature frequency effects in recognition memory. *Memory & Cognition*, 30(4):607–613, 2002.

891. B. C. Malt and S. A. Sloman. Linguistic diversity and object naming by non-native speakers of English. *Bilingualism: Language and Cognition*, 6(1):47–67, 2003.

892. B. C. Malt, S. A. Sloman, S. Gennari, M. Shi, and Y. Wang. Knowing versus naming: Similarity and the linguistic categorization of artifacts. *Journal of Memory and Language*, 40:230–262, 1999.

893. B. C. Malt, S. A. Sloman, and S. P. Gennari. Universality and language specificity in object naming. *Journal of Memory and Language*, 49(1):20–42, 2003.

894. J. M. Mandler and N. S. Johnson. Remembrance of things parsed: Story structure and recall. *Cognitive Psychology*, 9:111–151, 1977.

895. A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the* 16^{th} *IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

896. A. B. Markman and E. J. Wisniewski. Similar and different: The differentiation of basic-level categories. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 23(1):54–70, 1997.

897. A. I. Markushevich. *Theory of Functions of a Complex Variable*. American Mathematical Society, second edition, 1998.

898. M. Martin. Memory span as a measure of individual differences in memory capacity. *Memory & Cognition*, 6(2):194–198, 1978.

899. M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 125–135. IEEE, Dec. 1997.

900. W. J. Masek and M. S. Paterson. How to compute string-edit distances quickly. In D. Sankoff and J. Kruskal, editors, *Time Warps, String Edits, and Macromolecules*, chapter 14, pages 337–349. CSLI Publications, 1999.

901. H. Massalin. Superoptimizer – A look at the smallest program. In *Second International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS II), pages 122–126. ACM Press, Oct. 1987.

902. D. W. Massaro, R. L. Venezky, and G. A. Taylor. Orthographic regularity, positional frequency, and visual processing of letter strings. *Journal of Experimental Psychology: General*, 108(1):107–122, 1979.

903. E. Matias, I. S. MacKenzie, and W. Buxton. One-handed touchtyping on a QWERTY keyboard. *Human-Computer Interaction*, 11:1–27, 1996.

904. D. W. Matula. In and out conversions. *Communications of the ACM*, 11:47–50, 1968.

905. R. E. Mayer. Qualitatively different encoding strategies for linear reasoning premises: Evidence for single association and distance theories. *Journal of Experimental Psychology: Human Learning and Memory*, 5(1):1–10, 1979.

906. A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS-VI: Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct. 1994.

907. J. Mayrand, C. Leblanc, and E. M. Merlo. Automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254. IEEE Computer Society Press, Nov. 1996.

908. A. G. R. McClelland, R. E. Rawles, and F. E. Sinclair. The effects of search criteria and retrieval cue availability on memory for words. *Memory & Cognition*, 9(2):164–168, 1981.

909. D. C. McClelland. Testing for competence rather than for "intelligence". *American Psychologist*, 28:1–14, Jan. 1973.

910. J. L. McClelland and D. E. Rumelhart. An interactive activation model of context effects in letter perception: Part 1. An account of basic findings. *Psychological Review*, 88(5):375–405, 1981.

911. M. McCloskey, A. Washburn, and L. Felch. Intuitive physics: The straight-down belief and its origin. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(4):636–649, 1983.

912. S. McConnell. *Code Complete*. Microsoft Press, 1993.

913. K. B. McDermott. The persistence of false memories in list recall. *Journal of Memory and Language*, 35:212–230, 1996.

914. D. McFadden. Rationality for economists? *Journal of Risk and Uncertainty*, 19:73–105, 1999.

915. D. C. McFarlane. Interruption of people in human-computer interaction: A general unifying definition of human interruption and taxonomy. Technical Report NRL/FR/5510-97-9870, Naval Research Laboratory, Dec. 1997.

916. S. McFarling. Procedure merging with instruction caches. Technical Report WRL Research Report 91/5, Digital Western Research Laboratory, Mar. 1991.

917. K. B. McKeithen, J. S. Reitman, H. H. Ruster, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

918. K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.

919. E. McKone. Short-term implicit memory for words and nonwords. *Journal of Experimental Psychology: Learning, Memory & Cognition*, 21(5):1108–1126, 1995.

920. G. McKoon and R. Ratcliff. Contextually relevant aspects of meaning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(2):331–343, 1988.

921. L. McMahan and R. Lee. Pathlengths of SPEC benchmarks for PA-RISC, MIPS, and SPARC. In *Proceedings of IEEE Compcon*, pages 481–490, Feb. 1993.

922. J. McMillan. Enhancing college student's critical thinking: A review of studies. *Research in Higher Education*, 26:3–29, 1987.

923. T. P. McNamara, J. K. Hardy, and S. C. Hirtle. Subjective hierarchies in spatial memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(2):211–227, 1989.

924. K. H. McWeeny, A. W. Young, D. C. Hay, and A. W. Ellis. Putting names to faces. *British Journal of Psychology*, 78:143–149, 1987.

925. R. E. Melchers and M. V. Harrington. Human error in simple design tasks. Technical Report Civil Engineering Research Reports Report Number 31, Monash University, 1982.

926. R. C. Merkle. Energy limits to the computational power of the human brain. *Foresight Update*, 6, Aug. 1989.

927. Metroworks. *CodeWarrior C Compilers Reference*. Metroworks Corp., Aug. 2001.

928. S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley professional computing series. Addison–Wesley, Reading, MA, USA, 1992.

929. S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, Rhode Island 02912, U.S.A., Apr. 1993.

930. S. Meyers and M. Lejter. Automatic detection of C++ programming errors: Initial thoughts on a lint++. Technical Report Technical Report CS-91-51, Brown University, 1991.

931. Microchip. *PIC18CXX2 High-Performance Microcontrollers with 10-Bit A/D*, ds39026b edition, 1999.

932. Microsoft. *Microsoft QuickC Compiler*. Microsoft Corporation, 1987.

933. A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 155–162, Oct. 2002.

934. S. Milgram. *Obedience to Authority*. McGraw–Hill, 1974.

935. G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.

936. G. A. Miller, J. S. Bruner, and L. Postman. Familiarity of letter sequences and tachistoscope identification. *The Journal of General Psychology*, 50:129–139, 1954.

937. G. A. Miller and S. Isard. Free recall of self-embedded English sentences. *Information and Control*, 7:292–303, 1964.

938. J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research. Technical Report EFOCS-15-94, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1994.

939. J. Miller, M. Wood, M. Roper, and A. Brooks. Further experiences with scenarios and checklists. Technical Report EFOCS-20-94, University of Strathclyde, 1994.

940. J. S. Miller and G. J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report A.I. Memo No. 1462, M.I.T., Mar. 1994.

941. MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 1998.

942. MISRA. *MISRA-C:2004 Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2004.

943. MISRA. *MISRA-C++:2008 Guidelines for the use of the C++ language in critical systems*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2008.

944. B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measures. In *Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01)*, pages 744–753. IEEE, Apr. 2001.

945. S. Mithen. *The Prehistory of the Mind*. Thames and Hudson, 1996.

946. R. Mitton. *English Spelling and the Computer*. Longman, 1996.

947. A. Miyake and P. Shah. *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. Cambridge University Press, 1999.

948. E. Mizukami. The accuracy of floating-point summation for cg-like methods. Technical Report Technical Report 486, Indiana University, July 1997.

949. M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analysis and potential applications in program understanding and optimization. Technical Report UW CSE Technical Report 01-03-01, University of Washington, Mar. 2001.

950. M. U. Mock. *Automatic Selective Dynamic Compilation*. PhD thesis, University of Washington, 2002.

951. A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. Technical Report ALR-2002-003, Avaya Labs Research, Jan. 2002.

952. A. Mockus and D. M. Weiss. Predicting risk in software changes. *Bell Labs Technical Journal*, Apr.-June 2000.

953. T. Moher and G. M. Schneider. Methods for improving controlled experimentation in software engineering. In *Proceedings of the 5^{th} international conference on Software engineering*, pages 224–233. IEEE Computer Society, Mar. 1981.

954. C. Molina, A. González, and J. Tubella. Dynamic removal of redundant computations. In *Proceedings of the 13^{th} International Conference on Supercomputing*, pages 474–481. ACM Press, 1999.

955. J. Monaghan and A. W. Ellis. What exactly interacts with spelling-sound consistency in word naming? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 28(1):183–206, 2002.

956. P. Monaghan. A corpus-based analysis of individual differences in proof-style. Thesis (m.s.), Centre for Cognitive Science, University of Edinburgh, 1995.

957. P. Monaghan. *Representation and Strategy in Reasoning: An Individual Differences Approach*. PhD thesis, University of Edinburgh, 2000.

958. A. F. Monk and C. Hulme. Errors in proofreading: Evidence for the use of word shape in word recognition. *Memory & Cognition*, 11(1):16–23, 1983.

959. S. Monsell. Task switching. *TRENDS in Cognitive Science*, 7(3):134–140, 2003.

960. S. Monsell, K. E. Patterson, A. Graham, C. H. Hughes, and R. Milroy. Lexical and sublexical translation of spelling to sound: Strategic anticipation of lexical status. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 18(3):452–467, 1992.

961. J. E. Moore and L. A. Burke. How to turn around 'turnover culture' in IT. *Communications of the ACM*, 45(2):73–78, 2002.

962. J. A. Moorer. 48-bit integer processing beats 32-bit floating point for professional audio applications. In *AES 107^{th} Convention*, Sept. 1999. preprint 5038.

963. B. J. T. Morgan. Cluster analysis of two acoustic confusion matrices. *Perception & Psychophysics*, 13:12–24, 1973.

964. B. J. T. Morgan, S. M. Chambers, and J. Morton. Acoustic confusion of digits in memory and recognition. *Perception & Psychophysics*, 14(2):375–383, 1973.

965. J. Morris and G. Hirst. Lexical cohension computed by thesaural relations as an indicator of the structure of text. *Computational Linguistics*, 17(1):21–48, 1991.

966. D. Moseley. How lack of confidence in spelling affects children's written expression. *Educational Psychology in Practice*, 5(1):42–46, 1989.

967. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??

968. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.

969. Motorola, Inc. *PowerPC 601 RISC Microprocessor User's Manual*. Motorola, Inc, 1993.

970. Motorola, Inc. *DSP5600 24-bit Digital Signal Processor Family Manual*. Motorola, Inc, Austin, TX, USA, dsp56kfamum/ad edition, 1995.

971. Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.

972. Motorola, Inc. *SC140 DSP Core Reference Manual*. Motorola, Inc, 2 edition, Apr. 2001.

973. T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the USENIX 2^{nd} Symposium on Operating Systems Designed and Implementation*, pages 3–17. USENIX Association, Oct. 1996.

974. R. S. Moyer and R. H. Bayer. Mental comparison and the symbolic distance effect. *Cognitive Psychology*, 8:228–246, 1976.

975. R. S. Moyer and T. K. Landauer. Time required for judgements of numerical inequality. *Nature*, 215:1519–1520, 1967.

976. S. S. Muchnick. *Advances Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

977. S. T. Mueller, T. L. Seymour, D. E. Kieras, and D. E. Meyer. Theoretical implications of articulatory duration, phonological similarity, and phonological complexity in verbal working memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 29(6):1353–1380, 2003.

978. S. T. Mueller, C. T. Weidemann, and R. M. Shiffrin. Alphabetic letter similarity matrices: Effects of bias, perceivability, similarity and evidence discounting. Sept. 2007.

979. J.-M. Muller. On the definition of ulp (x). Technical Report No. 5504, Unité de recherche INRIA Rhône-Alpes, Feb. 2005.

980. M. Müller-Olm and O. Rüthing. On the complexity of constant propagation. In D. Sands, editor, 10^{th} *European Symposium on Programming (ESCOP 2001)*, pages 190–205. Springer-Verlag, Apr. 2001.

981. G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, Apr. 1998.

982. G. C. Murphy, D. Notkin, and E. S. C. Lan. An empirical study of static call graph extractors. In *Proceedings of the* 18^{th} *International Conference on Software Engineering*, pages 90–99. IEEE Computer Society Press/ACM Press, 1996.

983. G. L. Murphy and D. L. Medin. The role of theories in conceptual coherence. *Psychological Review*, 92(3):289–315, 1985.

984. P. Muter and E. E. Johns. Learning logographies and alphabetic codes. *Applied Cognitive Psychology*, 4:105–125, 1985.

985. R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, The Department of Computer Science, University of Arizona, Wednesday, Dec. 9 1998.

986. R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. In *Proceedings* 7^{th} *International Static Analysis Symposium (SAS 2000)*, volume 1824 of *LNCS*, pages 340–359. Springer, 2000.

987. I. B. Myers, M. H. McCaulley, N. L. Quenk, and A. L. Hammer. *A Guide to the Development and Use of the Myers-Briggs Type Indicator*. Consulting Psychologists Press, third edition, 1998.

988. C. R. Mynatt, M. E. Doherty, and W. Dragan. Information relevance, working memory, and the consideration of alternatives. *Quarterly Journal of Experimental Psychology*, 46A(4):759–778, 1993.

989. C. R. Mynatt, M. E. Doherty, and R. D. Tweney. Confirmation bias in a simulated research environment. *Quarterly Journal of Experimental Psychology*, 29:85–95, 1997.

990. R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the* 34^{th} *Annual International Symposium on Microarchitecture*, pages 40–51, Dec. 2001.

991. J. S. Nairne. A feature model of immediate memory. *Memory & Cognition*, 18(3):251–269, 1990.

992. T. Nakra. *A framework for performing prediction in VLIW architectures*. PhD thesis, University of Pittsburgh, 2001.

993. J. Nandigam. *A Measure for Module Cohesion*. PhD thesis, University of Louisiana, May 1995.

994. NASA. NASA GB-1740.13-96: NASA guidebook for safety critical software - analysis and development. Technical report, NASA Glenn Research Center, 1996.

995. P. Nation and R. Waring. Vocabulary size, text coverage and word lists. In N. Schmitt and M. McCarthy, editors, *Vocabulary: Description, Acquisition and Pedagogy*, chapter 1.1, pages 6–19. Cambridge University Press, 1998.

996. G. Navarro, R. Baeza-Yates, and J. ao Marcelo Azevedo Arcoverde. Matchsimile: A flexible approximate matching tool for personal names searching. *Journal of the American Society of Information Systems and Technology*, 54(1):3–15, 2003.

997. I. Neamtiu. Detailed break-down of general data provided in paper[998] kindly supplied by first author. Jan. 2008.

998. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.

999. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, June 2006.

1000. I. Neath and J. S. Nairne. Word-length effects in immediate memory: Overwriting trace decay theory. *Psychonomic Bulletin & Review*, 2(4):429–441, 1995.

1001. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.

1002. G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95, 2000.

1003. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

1004. G. C. Necula, S. McPeak, and W. Weimer. Taming C pointers. www.cs.berkeley.edu/~necula, 2004.

1005. D. L. Nelson and C. McEvoy. Word fragments as retrieval cues: Letter generation or search through nonsemantic memory? *American Journal of Psychology*, 97(1):17–36, 1984.

1006. D. L. Nelson, C. L. McEvoy, and T. A. Schreiber. The university of south Florida word association, rhyme and word fragment norms. Technical report, University of South Florida, Aug. 1999. www.usf.edu/FreeAssociation.

1007. K. M. Nelson, H. J. Nelson, and M. Ghods. Understanding the personal competencies of IS support experts: Moving towards the E-business future. In 34^{th} *Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 8*. IEEE, Jan. 2001.

1008. A. Newell and P. S. Rosenbloom. Mechanisms of skill acquisition and the power law of practice. In J. R. Anderson, editor, *Cognitive skills and their acquisition*, pages 1–54. Erlbaum, Hillsdale, NJ, 1981.

1009. K. C. Ng and FP group of SunPro. Argument reduction for huge arguments: Good to the last bit. Work in progress, Mar. 1992.

1010. D. M. Nichols and M. B. Twidale. Usability and open source software. Technical Report Working Paper 10/02, University of Waikato, 2002.

1011. R. S. Nickerson, D. N. Perkins, and E. E. Smith. *The Teaching of Thinking*. Erlbaum, Hillsdale NJ, 1985.

1012. X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *IEEE Workshop on Signal Processing Systems (SiPS'99)*, 1999.

1013. B. K. Nirmal. *PROGRAMMING STANDARDS and GUIDELINES: COBOL edition*. Prentice-Hall, Inc, 1987.

1014. R. E. Nisbett, D. H. Krantz, C. Jepson, and Z. Kunda. The use of statistical heuristics in everyday inductive reasoning. *Psychological Review*, 90(4):339–363, 1983.

1015. R. E. Nisbett and A. Norenzayan. Culture and cognition. In D. Medin and H. Pashler, editors, *Stevens' Handbook of Experimental Psychology, Volume Two: Memory and Cognitive Processes*, chapter 13. John Wiley & Sons, third edition, Apr. 2002.

1016. M. Norrish. *C formalized in HOL*. PhD thesis, Cambridge University, 1998.

1017. R. M. Nosofsky. Exemplar-based accounts of relations between classification, recognition and typicality. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(4):700–708, 1988.

1018. I. Noveck, G. Chierchia, and E. Sylvestre. Linguistic-pragmatic factors in interpreting disjunctions. *Thinking and Reasoning*, 8(4):297–326, 2002.

1019. Numerical C Extensions Group. Float-point C extensions. Technical Report X3J11.1/93-028, American National Standards Institute, Aug. 1993.

1020. J. M. Nuttin Jr. Affective consequence of mere ownership: The name letter effect in twelve European languages. *European Journal of Social Psychology*, 17:381–402, 1987.

1021. M. Oaksford and N. Chater. A rational analysis of the selection task as optimal data selection. *Psychological Review*, 101(4):608–631, 1994.

1022. K. Oberauer, H.-M. Süß, O. Wilhelm, and W. W. Wittmann. The multiple faces of working memory: Storage, processing, supervision, and coordination. *Intelligence*, 31:167–193, 2003.

1023. J. Oberlander, R. Cox, P. Monaghan, K. Stenning, and R. Tobin. Individual differences in proof structures following multimodal logic teaching. In *Proceedings of the* 18^{th} *Annual Meeting of the Cognitive Science Society*, pages 201–206, 1996.

1024. S. F. Oberman. Design issues in high performance floating point arithmetic units. Technical Report CSL-TR-96-711, Stanford University, Dec. 1996.

1025. S. M. O'Donnell. *Programming for the World*. Prentice Hall, Inc, 1994.

1026. U. M. of Defence. *Defence Standard 00-55. Requirements for safety related software in defence equipment. Part 2: Guidance*. UK Ministry of Defence, Aug. 1997.

1027. A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, 1993.

1028. K. Oflazer and C. Güzet. Spelling correction in agglutinative languages. Technical Report BU-CEIS-9401, Bilkent University, 1994.

1029. K. O'Hara. Towards a typology of reading goals. Technical Report Technical Report EPC-1996-107, Rank Xerox Research Centre, 1996.

1030. M. C. Ohlsson. Utilisation of historical data for controlling and improving software development. Licentiate, Lund Institute of Technology, Sweden, 1999.

1031. M. C. Ohlsson. *Controlling Fault-Prone Components for Software Evolution*. PhD thesis, Lund Institute of Technology, Sweden, 2001.

1032. Y. Oiwa, T. Sekiguichi, E. Sumii, and A. Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems*, pages 133–153. Springer-Verlag, Apr. 2003.

1033. T. Okada. A corpus analysis of spelling errors made by Japanese EFL writers. *Yamagata English Studies*, 9:17–36, 2004.

1034. T. Okada. A corpus-based study of spelling errors of Japanese EFL writers with reference to errors occurring in word-initial and word-final positions. In V. Cook and B. Basetti, editors, *Second Language Writing Systems*, chapter 6, pages 164–183. Multilingual Matters Limited, 2005.

1035. A. R. Omondi. *Computer Arithmetic Systems: Algorithms, architecture, and implementation*. Prentice Hall, Inc, 1994.

1036. N. Osaka. Eye fixation and saccade during kana and kanji text reading: Comparison of English and Japanese text processing. *Bulletin of the Psychonomic Society*, 27(6):548–550, 1989.

1037. D. N. Osherson, O. Wilkie, E. Shafir, E. E. Smith, and A. López. Category-based induction. *Psychological review*, 97(2):185–200, 1990.

1038. K. R. Paap, S. L. Newsome, and R. W. Noel. Word shape's in poor shape for the race to the lexicon. *Journal of Experimental Psychology: Human Perception and Performance*, 10(3):413–428, 1984.

1039. H. Packard. *HP C/iX Reference Manual*. Hewlett Packardi, Inc, USA, 3 edition, Apr. 1999.

1040. H. Packard. *HP C/HP-UX Programmer's Guide*. Hewlett Packard Company, tenth edition, Dec. 2001.

1041. P. W. Paese and J. A. Sniezek. Influences on the appropriateness of confidence in judgment: Practice, effort, information, and decision-making. *Organizational Behavior and Human Decision Processes*, 48:100–130, 1991.

1042. V. Pagel, K. Lenzo, and A. W. Black. Letter to sound rules for accented lexicon compression. In *ICSLP98*, volume 5, pages 2015–2020, 1998.

1043. K. V. Palem and R. M. Rabbah. Bridging processor and memory performance in ILP processors via data-remapping. Technical Report GIT-CC-01-014, Georgia Institute of Technology, June 2001.

1044. J. Palm and J. E. B. Moss. When to use a compilation service? In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems, LCTES'02*, pages 194–203. ACM, June 2002.

1045. D. D. Palmer. A trainable rule-based algorithm for word segmentation. In P. R. Cohen and W. Wahlster, editors, *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 321–328. Association for Computational Linguistics, 1997.

1046. J. Palmer, P. Verghese, and M. Pavel. The psychophysics of visual search. *Vision Research*, 40:1227–1268, 2000.

1047. S. E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, 1999.

1048. V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 303–314, Mar. 2004.

1049. P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P. G. K. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation and Electronic Systems*, 6(2):149–206, Apr. 2001.

1050. P. R. Panda, N. D. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384–409, Oct. 1997.

1051. S. Paoli. C++ coding standard specification. Technical Report CERN-UCO/1999/207, CERN, Jan. 2000.

1052. N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Greece, Feb. 1998.

1053. D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *Proceedings of Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 233–244, Feb. 2002.

1054. R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Sept. 1992.

1055. J. R. Parker. A general character to integer conversion method. *Software–Practice and Experience*, 15(8):761–766, 1985.

1056. J. M. Parkman. Temporal aspects of simple multiplication and comparison. *Journal of Experimental Psychology*, 95(2):437–444, 1972.

1057. J. M. Parkman and G. J. Groen. Temporal aspects of simple addition and comparison. *Journal of Experimental Psychology*, 89(2):335–342, 1971.

1058. M. Parks. Number-theoretic test generation for directed rounding. *IEEE Transactions on Computers*, 49(2):651–658, July 2000.

1059. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

1060. H. E. Pashler. *The Psychology of Attention*. The MIT Press, 1999.

1061. M. A. Paskin. Maximum entropy probabilistic logic. Technical report, University of California, Berkeley, USA, 2002.

1062. A. Patel. Auditors' belief revision: Recency effects of contrary and supporting audit evidence and source reliability. *The Auditors Report*, 24(3), 2001.

1063. E. Paulesu, E. McCrory, F. Fazio, L. Menoncello, N. Brunswick, S. F. Cappa, M. Cotelli, G. Cossu, F. Corte, M. Lorusso, S. Pesenti, A. Gallagher, D. Perani, C. Price, C. D. Frith, and U. Frith. A cultural effect on brain function. *Nature Neuroscience*, 3(1):91–96, Jan. 2000.

1064. A. Pavese and C. Umiltà. Symbolic distance between numerosity and identity moulates stroop-like interference. *Journal of Experimental Psychology: Human Perception and Performance*, 24(5):1535–1545, 1998.

1065. J. W. Payne, J. R. Bettman, and E. J. Bettman. *The Adaptive Decision Maker*. Cambridge University Press, 1993.

1066. M. J. Pazzani. Influence of prior knowledge on concept acquisition: Experimental and computational results. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 17(3):416–432, 1991.

1067. J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

1068. R. Peereman and A. Content. Orthographic and phonological neighborhood in naming: Not all neighbors are equally influential in orthographic space. *Journal of Memory and Language*, 37:382–410, 1997.

1069. R. Peereman and A. Content. Quantitative analyses of orthography to phonology mapping in English and French. student.vub.ac.be/~acontent/OPMapping.html, 1998.

1070. D. G. Pelli, C. W. Burns, B. Farell, and D. C. Moore. Identifying letters. *Vision Research*, 46(28):4646–4674, 2006.

1071. K. Peng, D. R. Ames, and E. D. Knowles. Culture and human inference: Perspectives from three traditions. In D. Matsumoto, editor, *The Handbook of Culture and Psychology*, pages 243–263. Oxford University Press, Oct. 2001.

1072. N. Pennington. Comprehension strategies in programming. In G. Olson, S. Shepard, and E. Soloway, editors, *Empirical Studies of programmers: Second Workshop*, chapter 7, pages 100–113. Ablex Publishing, 1987.

1073. N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

1074. M. Perea and E. Rosa. Does "whole word shape" play a role in visual word recognition? *Perception and Psychophysics*, 64(5):785–794, 2002.

1075. Perkin-Elmer. *C PROGRAMMING Manual*. Perkin-Elmer Corporation, Oceanport, New Jersey 07757, 1984.

1076. C. Perry, J. C. Ziegler, and M. Coltheart. How predictable is spelling? Developing and testing metrics of phoneme-grapheme contingency. *The Quarterly Journal of Experimental Psychology*, 55A(3):897–915, 2002.

1077. D. E. Perry and W. M. Evangelist. An empirical study of software interface faults. In *International Symposium on New Directions in Computing*, pages 32–38. IEEE, Aug. 1985.

1078. D. E. Perry, N. A. Staudenmayer, and L. G. Votta Jr. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.

1079. D. E. Perry, N. A. Staudenmayer, and L. G. Votta Jr. Understanding and improving time usage in software development. In A. Fuggetta and A. L. Wolf, editors, *Trends in Software Process*, chapter 5. John Wiley & Sons, 1996.

1080. D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 1993 European Software Engineering Conference*, pages 48–67, 1993.

1081. J. L. Peterson. A note on undetected typing errors. *Communications of the ACM*, 29(7):633–637, 1986.

1082. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.

1083. C. Phillips. *Order and Structure*. PhD thesis, M.I.T., Aug. 1996.

1084. R. J. Phillips. Why is lower case better? *Applied Ergonomics*, 10(4):211–214, 1979.

1085. R. J. Phillips. Searching for a target in a random arrangement of names: An eye fixation analysis. *Canadian Journal of Psychology*, 35(4):330–346, 1981.

1086. R. Pieters and L. Warlop. Visual attention during brand choice: The impact of time pressure and task motivation. *International Journal of Research in Marketing*, 16:1–16, 1999.

1087. R. Pike. How to use the Plan 9 C compiler. In *Plan 9 Programmer's Manual*. AT&T Bell Laboratories, 1995.

1088. S. Pinker. *How the Mind Works*. Penguin, 1997.

1089. S. Pinker. *Words and Rules*. Weidenfeld & Nicolson, 1999.

1090. A. Pirkola. Morphological typology of languages in IR. *Journal of Documentation*, 57(3):330–348, 2001.

1091. P. Pirolli and S. K. Card. Information foraging. *Psychological Review*, 106(4):643–675, 1999.

1092. M. Plauché, C. Delogu, and J. J. Ohala. Asymmetries in consonant confusion. In *Proceedings of Eurospeech 1997: 5^{th} European Conference on Speech Communication and Technology: Vol 4*, pages 2187–2190, Sept. 1997.

1093. D. C. Plaut and J. R. Booth. Individual and developmental differences in semantic priming: Empirical and computational support for a single-mechanism account of lexical processing. *Psychological Review*, 107:786–823, 2000.

1094. D. C. Plaut, J. L. McClelland, M. S. Seidenberg, and K. Patterson. Understanding normal and impaired word reading: Computational principles in quasi-regular domains. *Psychological Review*, 103(1):56–115, 1996.

1095. R. Ploetzner and K. VanLehn. The acquisition of qualitative physics knowledge during textbook-based physics trainings. *Cognition and Instruction*, 15(2):169–205, 1997.

1096. T. Plum. *Reliable data structures in C*. Plum Hall, 1985.

1097. T. Plum. *C Programming guidelines*. Plum Hall, 1989.

1098. T. Plum and D. Saks. *C++ Programming Guidelines*. Plum Hall, 1991.

1099. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

1100. A. Pollatsek, S. Bolozky, A. D. Well, and K. Rayner. Asymmetries in the perceptual span for Israeli readers. *Brain and Language*, 14:174–180, 1981.

1101. T. Pollmann and C. Jansen. The language user as an arithmetician. *Cognition*, 59:219–237, 1996.

1102. J. J. Pollock and A. Zamora. Collection and characterization of spelling errors in scientific and scholarly text. *Journal of the American Society for Information Science*, 34(1):51–58, 1983.

1103. M. Popovic and P. Willett. The effectiveness of stemming for natural-language access to Slovene textual data. *Journal of the American Society for Information Science*, 43(5):384–390, 1992.

1104. K. R. Popper. *Conjectures and Refutations*. Routledge, 1969.

1105. A. Porter, H. Siy, and L. Votta. A review of software inspections. In M. Zelkowitz, editor, *Advances in Computers 42*, pages 39–76. Academic Press, 1996.

1106. A. Porter and L. Votta. Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical Software Engineering*, 3(4):355–379, 1998.

1107. A. A. Porter, H. Siy, A. Mockus, and L. G. Votta. Understanding the sources of variation in software inspections. Technical Report CS-TR-3762, University of Maryland, College Park, Jan. 1997.

1108. A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, 23(6):329–346, June 1997.

1109. POSC. *POSC Base Computer Standards: version 2*. Prentice Hall, Inc, 1994.

1110. M. Postiff, D. Greene, and T. Mudge. The need for large register files in integer codes. Technical Report CSE-TR-434-00, The University of Michigan, Aug. 2000.

1111. A. Postma, R. Izendoorn, and E. H. F. De Haan. Sex differences in object location memory. *Brain and Cognition*, 36:334–345, 1998.

1112. E. M. Pothos and N. Chater. Rational categories. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, pages 848–853, 1998.

1113. M. C. Potter, A. Moryadas, I. Abrams, and A. Noel. Word perception and misperception in context. *Journal of Experimental Psychology: Learning, Memory & Cognition*, 19(1):3–22, 1993.

1114. G. R. Potts. Storing and retrieving information about ordering relationships. *Journal of Experimental Psychology*, 103(3):431–439, 1974.

1115. D. M. W. Powers. Applications and explanations of Zipf's law. In D. M. W. Powers, editor, *NeMLaP3/CoNLL98: New Methods in Language Processing and Computational Natural Language Learning*, pages 151–160, 1998.

1116. L. Prechelt. Why we need an explicit forum for negative results. *Journal of Universal Computer Science*, 3(9):1074–1083, 1997.

1117. L. Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report iratr-1999-18, Universität Karlsruhe, 1999.

1118. L. Prechelt. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, Oct. 1999.

1119. L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl for a string processing program. Technical Report Technical Report 2000-5, Universität Karlsruhe, Fakultat für Informatik, 2000.

1120. L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report Technical Report 2000-1, Universität Karlsruhe, Fakultat für Informatik, 2000.

1121. L. Prechelt and W. F. Tichy. A controlled experiment measuring the effect of procedure argument type checking on programmer productivity. Technical Report Technical Report CMU/SEI-96-TR-014, Software Engineering Institute, Carnegie Mellon University, 1996.

1122. C. C. Presson and D. R. Montello. Updating after rotational and translational body movements: coordinate structure of perspective space. *Perception*, 23:1447–1455, 1994.

1123. D. M. Priest. *On Properties of Floating-Point Arithmetic: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, Nov. 1992.

1124. L. Pring. Phonological codes and functional spelling units: Reality and implications. *Perception and Psychophysics*, 30(6):573–578, 1981.

1125. T. A. Proebsting and B. G. Zorn. Programming shorthand. Technical Report Technical Report MSR-TR-2000-03, Microsoft Research, 2000.

1126. J. B. Proffitt, J. D. Coley, and D. L. Medin. Expertise and category-based induction. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 26(4):811–828, 2000.

1127. C. Pronk. Stress testing of compilers for Modula-2. *Software–Practice and Experience*, 22(10):885–897, 1992.

1128. PTSC. *Patriot C Development Tools Reference Manual*. Patriot Scientific Corporation, San Diego, CA, 1.1 edition, Feb. 2001.

1129. PTSC. *IGNITE Intellectual Property Reference Manual*. Patriot Scientific Corporation, San Diego, CA, 1.0 edition, Mar. 2002.

1130. Z. Pylyshyn. Is vision continuous with cognition? The case for cognitive impenetrability of visual perception. *Behavioral and Brain Sciences*, 22(3):341–423, 1999.

1131. S. Qualline. *C Elements of Style*. M&T Books, 1992.

1132. P. T. Quinlan and R. N. Wilton. Grouping by proximity or similarity? Competition between gestalt principles in vision. *Perception*, 27:417–430, 1998.

1133. G. Quinton and B. J. Fellows. 'Perceptual' strategies in the solving of three-term series problems. *British Journal of Psychology*, 66:69–78, 1975.

1134. M. Rabin and J. Schrag. First impressions matter: A model of confirmation bias. *Quarterly Journal of Economics*, 114:37–82, 1999.

1135. H. Rabinowitz and C. Schaap. *Portable C*. Prentice Hall, Inc, 1990.

1136. D. Raffo, J. Settle, and W. Harrison. Investigating financial measures for planning software IV&V. Technical Report TR-99-05, Portland State University, 1999.

1137. M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. System call clustering :A profile-directed optimization technique. www.cs.arizona.edu, May 2002.

1138. P. Åke Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the First International Symposium on Memory Management ISMM'98*, pages 176–185, 1998.

1139. R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and J. P. Shen. Performance advantage of the register stack in Intel Itanium processors. In 2^{nd} *Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, Nov. 2002.

1140. G. Ramalingam. Identifying loops in almost linear time. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, 1999.

1141. A. Ramírez, J.-L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. In *IEEE International Conference on Parallel Processing (ICPP99)*, 1999.

1142. J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill, Inc, 1992.

1143. P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

1144. A. Rao. Compiler optimizations for storage assignment on embedded DSPs. Thesis (m.s.), University of Cincinnati, Oct. 1998.

1145. K. Rastle and M. Coltheart. Whammies and double whammies: The effect of length on nonword reading. *Psychonomic Bulletin and Review*, 5:277–282, 1998.

1146. E. J. Ratliff. Decreasing process memory requirements by overlapping run-time stack data. Thesis (m.s.), Florida State University, College of Arts and Sciences, 1997.

1147. K. Rayner. Eye movements in reading and information processing: 20 years of research. *Psychology Bulletin*, 124(3):372–422, 1998.

1148. K. Rayner, K. S. Binder, J. Ashby, and A. Pollatsek. Eye movement control in reading: word predictability has little influence on initial landing position in words. *Vision Research*, 41:943–954, 2001.

1149. J. Reason. *Human Error*. Cambridge University Press, 1990.

1150. A. S. Reber and S. M. Kassin. On the relationship between implicit and explicit modes in the learning of a complex rule structure. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):492–502, 1980.

1151. RedHill Consulting, Pty. Simian - similarity analyser, v 2.0.2. www.redhillconsulting.com.au, 2004.

1152. G. M. Reicher. Perceptual recognition as a function of meaningfulness of stimulus material. *Journal of Experimental Psychology*, 81(2):275–280, 1969.

1153. E. D. Reichle, A. Pollatsek, D. L. Fisher, and K. Rayner. Towards a model of eye movement control in reading. *Psychological Review*, 105(1):125–157, 1998.

1154. E. D. Reichle, K. Rayner, and A. Pollatsek. The E-Z reader model of eye-movement control in reading: Comparison to other models. *Behavioral and Brain Sciences*, 26:445–526, 2003.

1155. D. J. Reifer. Qualifying the debate: Ada vs C++. *Crosstalk: The Journal of Defense Software Engineering*, 1996.

1156. T. W. Reps. "maximal-munch" tokenization in linear time. *ACM Transactions on Programming Languages and Systems*, 20(2):259–273, Mar. 1998.

1157. P. Research. *QAC Users Guide*. Programming Research, Hersham, Surrey, KT12 5LU, pc version 5.0 edition, Apr. 2003.

1158. P. Resnick. Semantic similarity in a taxonomy: An information based measure and its application to ambiguity in natural language. *Journal of A.I. Research*, 11:95–130, 1999.

1159. A. Rey, A. M. Jacobs, F. Schmidt-Weigand, and J. C. Ziegler. A phoneme effect in visual word recognition. *Cognition*, 68:B71–B80, 1998.

1160. A. Rey, J. C. Ziegler, and A. M. Jacobs. Graphemes are perceptual reading units. *Cognition*, 75:B1–B12, 2000.

1161. J. C. Richards. The role of vocabulary teaching. *TESOL Quarterly*, 10(1):77–89, 1976.

1162. M. Richards and C. Whitby-Strevens. *BCPL —the language and its compiler*. Cambridge University Press, 1979.

1163. J. Richardson and T. C. Ormerod. Rephrasing between disjunctives and conditionals: Mental models and the effects of thematic content. *The Quarterly Journal of Experimental Psychology*, 50A(2):358–385, 1997.

1164. S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, Inc, Sept. 1992.

1165. P. L. Richman. Floating-point number representations: base choice versus exponent range. Technical Report CS-TR-67-64, Stanford University, Department of Computer Science, Apr. 1967.

1166. L. J. Rips. Inductive judgments about natural categories. *Journal of Verbal Learning and Verbal Behavior*, 14:665–681, 1975.

1167. L. J. Rips. *The Psychology of Proof*. The MIT Press, 1994.

1168. L. J. Rips, E. J. Shoben, and E. E. Smith. Semantic distance and the verification of semantic relations. *Journal of Verbal Learning and Verbal Behavior*, 12:1–20, 1973.

1169. D. M. Ritchie. The development of the C language. *Second History of Programming Languages conference*, 1993.

1170. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. The C programming language. Technical Report 31, Bell Telephone Laboratories, Oct. 1975.

1171. T. G. Robertazzi and S. C. Schwartz. Best "ordering" for floating point addition. *ACM Transactions on Mathematical Software*, 14(1):101–110, 1988.

1172. E. S. Roberts. Implementing exceptions in C. Technical Report 40, Digital Systems Research Center, Mar. 1989.

1173. M. J. Roberts, D. J. Gilmore, and D. J. Wood. Individual differences and strategy selection in reasoning. *British Journal of Psychology*, 88:473–492, 1997.

1174. E. L. Robertson. Code generation and storage allocation for machines with span-dependent instructions. *ACM Transactions on Programming Languages and Systems*, 1(1):71–83, 1979.

1175. A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM Press, 2001.

1176. H. L. Roediger III, J. D. Jacoby, and K. B. McDermott. Misinformation effects in recall: Creating false memories through repeated retrieval. *Journal of Memory and Language*, 35:300–318, 1996.

1177. R. D. Rogers and S. Monsell. Costs of a predictable switch between simple cognitive tasks. *Journal of Experimental Psychology: General*, 124(2):207–231, 1995.

1178. R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, 2001.

1179. E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson, and P. Boyes-Braem. Basic objects in natural categories. *Cognitive Psychology*, 8:382–439, 1976.

1180. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.

1181. L. Ross, M. R. Lepper, and M. Hubbard. Perseverance in self-perception and social perception: Biased attributional processes in the debeliefing paradigm. *Journal of Personality and Social Psychology*, 32(5):880–892, 1975.

1182. E. Z. Rothkopf. Incidental memory for location of information in text. *Journal of Verbal Learning and Verbal Behavior*, 10:608–613, 1971.

1183. RTCA. Software considerations in airborne systems and equipment certifications, DOD-178B. Technical report, RTCA, Washington D.C., 1992.

1184. D. C. Rubin. Within word structure in the tip-of-the-tongue phenomenon. *Journal of Verbal Learning and Verbal Behavior*, 14:392–397, 1975.

1185. D. C. Rubin and A. E. Wenzel. One hundred years of forgetting: A quantitative description of retention. *Psychological Review*, 103(4):734–760, 1996.

1186. S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *The* 29^{th} *Annual ACM Symposium on Principles of Programming Languages, POPL'02*, pages 140–153, Jan. 2002.

1187. J. S. Rubinstein, D. E. Meyer, and J. E. Evans. Executive control of cognitive processes in task switching. *Journal of Experimental Psychology: Human Perception and Performance*, 27(4):763–797, 2001.

1188. D. E. Rumelhart and D. A. Norman. Similating a skilled typist: A study of skilled cognitive-motor performance. *Cognitive Science*, 6:1–36, 1982.

1189. J. Runeson. Code compression through procedure abstraction before register allocation. Thesis (m.s.), Uppsala University, Box 311, S-751 05 Uppsala, Sweden, 2000.

1190. R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report USC-CS-92-524, University of California, Berkeley, Sept. 1992.

1191. J. Sajaniemi. Visualizing roles of variables to novice programmers. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *Fourteenth Annual Workshop of the Psychology of Programming Interest Group*, pages 111–127, June 2002.

1192. T. A. Salthouse. The processing-speed theory of adult age differences in cognition. *Psychological Review*, 103(3):403–428, 1996.

1193. D. D. Salvucci. An integrated model of eye movements and visual encoding. *Cognitive Systems Research*, 1(4):201–220, 2001.

1194. G. Sampson. *Writing Systems: A Linguistic Introduction*. Stanford University Press, 1985.

1195. R. Samuels, S. Stich, and L. Faucher. Reason and rationality. In I. Niiniluoto, M. Sintonen, and J. Wolenski, editors, *Handbook of Epistemology*, pages 131–179. Dordrecht:Kluwer, 2004.

1196. T. Sanocki. Font regularity constraints on the process of letter recognition. *Journal of Experimental Psychology: Human Perception and Performance*, 14(3):472–480, 1988.

1197. W. Scacchi. Understanding software productivity. In D. Hurley, editor, *Advances in Software Engineering and Knowledge Engineering*, volume 4, pages 37–70. World Scientific, 1995.

1198. D. J. Scales. Efficient dynamic procedure placement. Technical Report Research Report 98/5, Compaq Western Research Laboratory, 1998.

1199. R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

1200. D. A. Schkade and D. N. Kleinmuntz. Information displays and choice processes: Differential effects of organization, form, and sequence. *Organizational Behavior and Human Decision Processes*, 57:319–337, 1994.

1201. I. Schloss. Chickens and pickles. *Journal of Advertising Research*, 21(6):47–49, Dec. 1981.

1202. W. Schneider. Training high-performance skills: Fallacies and guidelines. *Human Factors*, 27(3):285–300, 1985.

1203. J. W. Schoonard and S. J. Boies. Short type: A behavior analysis of typing and text entry. *Human Factors*, 17(2):203–214, 1975.

1204. N. L. Schryer. A test of a computer's floating-point arithmetic unit. Technical Report Computing Science Technical Report No. 89, Bell Laboratories, Murray Hill, NJ, USA, Feb. 1981.

1205. M. J. Schulte, K. E. Wires, and J. E. Stine. Variable-correction truncated floating point multipliers. In *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, pages 1344–1348. IEEE, 2000.

1206. C. D. Schunn, L. M. Reder, A. Nhouyvanisvong, D. R. Richards, and P. J. Stroffolino. To calculate or not to calculate: A source activation confusion model of problem familiarity's role in strategy selection. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(1):3–29, 1997.

1207. E. M. Schwarz and C. A. Krygowski. The S/390 G5 floating point unit. *IBM Journal of Research and Development*, 43(5/6):707–721, Sept.-Nov. 1999.

1208. A. Scott. The vertical direction and time in Mandarin. *Australian Journal of Linguistics*, 9:295–314, 1989.

1209. S. Scribner. Modes of thinking and ways of speaking: culture and logic reconsidered. In P. N. Johnson-Laird and P. C. Wason, editors, *Thinking: Readings in Cognitive Science*, chapter 29, pages 483–500. Cambridge University Press, 1977.

1210. P. Sedlmeier, R. Hertwig, and G. Gigerenzer. Are judgments of the positional frequencies of letters systematically biased due to availability? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 24(3):754–770, 1998.

1211. M. L. Seidl and B. G. Zorn. Implementing heap-object behavior prediction efficiently and effectively. *Software–Practice and Experience*, 31(9):869–892, 2001.

1212. R. W. Selby and V. R. Basili. Analysing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.

1213. E. O. Selkirk. *The Syntax of Words*. MIT Press, 1982.

1214. R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

1215. C. A. Sevald and G. S. Dell. The sequential cuing effect in speech production. *Cognition*, 53:91–127, 1994.

1216. L. H. Shaffer and J. Hardwick. Typing performance as a function of text. *Quarterly Journal of Experimental Psychology*, 20:360–369, 1968.

1217. T. M. Shaft and I. Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *Journal of Management Information Systems*, 15(1):51–78, 1998.

1218. R. C. Shank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding: An Enquiry into Human Knowledge Structures*. Lawrence Erlbaum Associates, 1977.

1219. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In 10^{th} *USENIX Security Symposium*, Aug. 2001.

1220. C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

1221. D. J. Shaw. A phonological interpretation of two acoustic confusion matrices. *Perception & Psychophysics*, 17(6):537–542, 1975.

1222. B. A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, Mar. 1981.

1223. J. W. Sheldon. Strength reduction of integer division and modulo operations. Thesis (m.s.), MIT, May 2001.

1224. Z. Shen, Z. Li, and P.-C. Yew. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990. CSRD TR#983.

1225. R. N. Shepard, C. I. Hovland, and H. M. Jenkins. Learning and memorization of classifications. *Psychological Monographs: General and Applied*, 75(15):1–39, 1961.

1226. R. N. Shepard and J. Metzler. Mental rotation of three-dimensional objects. *Science*, 171:701–703, Feb. 1971.

1227. T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report CS99-630, University of California, San Diego, Aug. 1999.

1228. R. M. Shiffrin and M. Steyvers. A model for recognition memory: REM – retrieving effectively from memory. *Psychonomic Bulletin & Review*, 4(2):145–166, 1997.

1229. R. J. Shiller. *Irrational Exuberance*. Princeton University Press, 2000.

1230. B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Inc, 1980.

1231. S. M. Shugan. The cost of thinking. *Journal of Consumer Research*, 7:99–111, Sept. 1980.

1232. L. J. Shustek. *Analysis and performance of computer instruction sets*. PhD thesis, Stanford University, Jan. 1978.

1233. A. Sides, D. Osherson, N. Bonini, and R. Viale. On the reality of the conjunction fallacy. *Memory & Cognition*, 30(2):191–198, 2002.

1234. J. G. Siek, J. M. Squyres, and A. Lumsdaine. The laboratory for scientific computing (LSC): Coding standards. Technical report, University of Notre Dame, Apr. 2000.

1235. M. B. Siff. *Techniques for software renovation*. PhD thesis, University of Wisconsin–Madison, 1998.

1236. S. Silberman. The geek syndrome. *Wired*, 9(12), Dec. 2001.

1237. Silicon Graphics. *C Language Reference Manual*. Silicon Graphics, Inc, 007-0701-130 edition, 1999.

1238. S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 180–187, June 1998.

1239. S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the Twentieth International Conference on Software Engineering*, pages 361–370, Apr. 1998.

1240. H. A. Simon. *Models of Bounded Rationality: Behavioral Economics and Business Organization*. The MIT Press, 1982.

1241. J.-M. Simon, K. E. Emam, S. Rousseau, E. Jacquet, and F. Babey. The reliability of ISO/IEC PDTR 15504 assessments. Technical Report Technical Report ISERN-97-28, Fraunhofer Institute for Experimental Software Engineering, 1997.

1242. I. Simonson. Choice based on reasons: The case of attraction and compromise effects. *Journal of Consumer Research*, 16:158–173, Sept. 1989.

1243. I. Simonson and A. Tversky. Choice in context: Tradeoff contrast and extremeness aversion. *Journal of Marketing Research*, 29:281–295, 1992.

1244. C. Simonyi. *Meta-Programming: A Software Production Method*. PhD thesis, Stanford University, 1977. msdn.microsoft.com/library/techart/hunganotat.htm.

1245. M. Singer and K. F. M. Ritchot. The role of working memory capacity and knowledge access in text inference processing. *Memory & Cognition*, 24(6):733–743, 1996.

1246. W. T. Siok and P. Fletcher. The role of phonological awareness and visual-orthographic skills in Chinese reading acquisition. *Developmental Psychology*, 37(6):886–899, 2001.

1247. R. L. Sites and R. L. Witek. *Alpha AXP architecture reference manual*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1995.

1248. D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, E. Karahasanovic, E. F. Koren, and M. Vokác. Conducting realistic experiments in software engineering. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, pages 17–26, Oct. 2002.

1249. D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. Technical Report 2004-4, SIMULA Research Laboratory, 2004.

1250. J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of CASES'01*, pages 15–23, Nov. 2001.

1251. K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the* 31^{st} *Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 259–271, Los Alamitos, Nov. 30–Dec. 2 1998. IEEE Computer Society.

1252. V. Skirbekk. Age and individual productivity: A literature survey. Technical Report MPIDR Working Paper WP 2003-028, Max Plank Institute for Demographic Research, Aug. 2003.

1253. N. J. Slamecka and P. Graf. The generation effect: Delineation of a phenomenon. *Journal of Experimental Psychology: Human Learning and Memory*, 4(6):592–604, 1978.

1254. N. T. Slingerland and A. J. Smith. Measuring the performance of multimedia instruction sets. Technical Report UCB/CSD-00-1125, University of California Berkeley, USA, Dec. 2000.

1255. J. A. Sloboda. Visual imagery and individual differences in spelling. In U. Frith, editor, *Cognitive Processes in Spelling*, chapter 11, pages 231–248. Academic Press, 1980.

1256. S. Sloman and D. A. Lagnado. Counterfactual undoing in deterministic causal reasoning. In *Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society*, 2002.

1257. S. A. Sloman. The empirical case for two systems of reasoning. *Psychological Bulletin*, 119(1):3–22, 1996.

1258. S. A. Sloman, M. C. Harrison, and B. C. Malt. Recent exposure affects artifact naming. *Memory & Cognition*, 30(5):687–695, 2002.

1259. M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, 1989.

1260. P. Smith. Implicit pointer conversion involving const and volatile. Technical Report WG21/N0283, JTC1/SC22/WG21 C++ Standards Committee, 1993.

1261. G. Snelting. Reengineering of configurations based on mathematical concept analysis. Technical Report Informatik-Bericht Nr. 95-02, Technische Universität Braunschweig, Jan. 1995.

1262. B. So and M. W. Hall. Increasing the applicability of scalar replacement. In *Compiler Construction, 13th International Conference, CC 2004*, pages 185–201, Mar. 2004.

1263. P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square root implementations. *Computing Surveys*, 28(3):518–564, Sept. 1996.

1264. S. Sokolova and D. R. Kaeli. Static branch prediction using high-level language. Technical Report Technical Report ECE-CEG-98-009, Northeastern University, 1998.

1265. R. L. Solso and P. F. Barabuto jr. Bigram and trigram frequencies and versatilities in the English language. *Behavior Research Methods & Instrumentation*, 11(5):475–484, 1979.

1266. R. L. Solso and C. L. Juel. Positional frequency and versatility of bigrams for two- through nine-letter English words. *Behavior Research Methods & Instrumentation*, 12(3):297–343, 1980.

1267. R. L. Solso and J. F. King. Frequency and versatility of letters in the English language. *Behavior Research Methods & Instrumentation*, 8(3):283–286, 1976.

1268. S. Sonnentag. Excellent software professionals: experience, work activities, and perception by peers. *Behaviour & Information Technology*, 14(5):289–299, 1995.

1269. F. Spadini, M. Fertig, and S. J. Patel. Characterization of repeating dynamic code fragments. Technical Report CRHC-02-09, University of Illinois at Urbana-Champaign, 2002.

1270. A. Spector and I. Biederman. Mental set and shift revisited. *American Journal of Psychology*, 89:669–679, 1976.

1271. D. Sperber and D. Wilson. *Relevance: Communication and Cognition*. Blackwell Publishers, second edition, 1995.

1272. K. T. Spoehr and E. E. Smith. The role of orthographic and phonotactic rules in perceiving letter patterns. *Journal of Experimental Psychology: Human Perception and Performance*, 104(1):21–34, 1975.

1273. R. Sproat, A. Black, S. Chen, S. Kumar, M. Ostendorf, and C. Richards. Normalization of non-standard words: WS'99 final report. Technical Report www.clsp.jhu.edu/ws99/projects/normal, The Johns Hopkins University, Sept. 1999.

1274. D. Spuler. *C++ and C debugging, testing and reliability*. Prentice Hall, Inc, 1994.

1275. D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Technical Report Technical Report 94/03, James Cook University, Jan. 1994.

1276. A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. Technical Report Research Report 954/1, Western Research Laboratory - Compaq, Feb. 1994.

1277. S. Srivastava, M. Hicks, J. S. Foster, and P. Jenkins. Modular information hiding and type-safe linking for C. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 3–14, Apr. 2007.

1278. R. M. Stallman. *Using te GNU Compiler Collection*. Free Software Foundation, Mar. 2004.

1279. D. Stamovlasis and G. Tsaparlis. Non-linear analysis of the effect of working-memory capacity on organic-synthesis problem solving. *Chemistry Education: Research and Practice in Europe*, 1(3):375–380, 2000.

1280. L. Standing, J. Conezio, and R. N. Haber. Perception and memory for pictures: Single-trial learning of 2500 visual stimuli. *Psychonomic Science*, 19(2):73–74, 1970.

1281. Standish Group. The chaos report. Technical report, The Standish Group, 1995.

1282. J. Stanlaw. Two observations on culture contact and the Japanese color nomenclature system. In C. L. Hardin and L. Maffi, editors, *Color categories in thought and language*, chapter 11, pages 240–260. Cambridge University Press, 1997.

1283. K. E. Stanovich. *Who Is Rational?* Lawrence Erlbaum Associates, 1999.

1284. K. E. Stanovich and D. W. Bauer. Experiments on the spelling-to-sound regularity effects in word recognition. *Memory & Cognition*, 6(4):410–415, 1978.

1285. M. L. Staples and J. M. Bieman. 3-D visualization of software structure. In M. Zelkowitz, editor, *Advances in Computers, Volume 49*, pages 96–143. Academic Press, Apr. 1999.

1286. G. L. Steele Jr. Arithmetic shifting considered harmful. Technical Report A.I. Memo No. 378, M.I.T., Sept. 1976.

1287. G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. In *Proceeding of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

1288. J. L. Steffen. Adding run-time checking to the portable C compiler. *Software–Practice and Experience*, 22(4):305–316, 1992.

1289. D. W. Stephens and J. R. Krebs. *Foraging Theory*. Princeton University Press, 1986.

1290. M. W. Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Thesis (m.s.), M.I.T, Cambridge, MA, USA, May 2000.

1291. C. M. Sterling. Spelling errors in context. *British Journal of Psychology*, 74:353–364, 1983.

1292. R. J. Sternberg. Representation and process in linear syllogistic reasoning. *Journal of Experimental Psychology: General*, 109(2):119–159, 1980.

1293. R. J. Sternberg and E. M. Weil. An aptitude × strategy interaction in linear syllogistic reasoning. *Journal of Educational Psychology*, 72(2):226–239, 1980.

1294. S. Sternberg. Memory-scanning: Mental processes revealed by reaction-time experiments. *American Scientist*, 57(4):421–457, 1969.

1295. A. Stevens and P. Coupe. Distortions in judged spatial relations. *Cognitive Psychology*, 10:422–437, 1978.

1296. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 38(2-3):231–256, 1999.

1297. T. R. Stewart and C. M. Lusk. Seven components of judgmental forecasting skill: Implications for research and improving forecasts. *Journal of Forecasting*, 13:579–599, 1994.

1298. M. Steyvers. *Modeling semantic and orthographic similarity effects on memory for individual words*. PhD thesis, Indiana University, Sept. 2000.

1299. M. Steyvers and K. J. Malmberg. The effect of normative contextual variability on recognition memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 29(5):760–766, 2003.

1300. M. Steyvers and J. B. Tenenbaum. The large-scale structure of semantic networks: Statistical analysis and a model of semantic growth. *Cognitive Science*, 29(1):41–78, Jan. 2005.

1301. M. Stiff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. Technical Report Technical Report BL0113590-990202-03, Bell Laboratories, 1999.

1302. M. B. Stiff. *Techniques for software renovation*. PhD thesis, University of Wisconsin-Madison, 1998.

1303. G. O. Stone, M. Vanhoy, and G. C. V. Orden. Perception is a two-way street: Feedforward and feedback phonology in visual word recognition. *Journal of Memory and Language*, 36:337–359, 1997.

1304. E. Strain, K. Patterson, and M. S. Seidenberg. Semantic effects in single-word naming. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(5):1140–1154, 1995.

1305. D. Straker. *C-Style standards and guidelines*. Prentice Hall, Inc, 1992.

1306. J. Strang, T. O'Reilly, and L. Mui. *Termcap and Terminfo*. O'Reilly & Associates, Inc, 1989.

1307. M. Strathern. 'Improving ratings': audit in the British university system. *European Review*, 5(3):305–321, 1997.

1308. L. A. Streeter, J. M. Ackroff, and G. A. Taylor. On abbreviating command names. *The Bell System Technical Journal*, 62(6):1807–1826, 1983.

1309. J. R. Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 28:643–662, 1935.

1310. B. Stroustrup. *The Design and Evolution of C++*. Addison–Wesley, 1999.

1311. C.-L. Su, C.-Y. Rsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. Technical Report ACAL-TR-94-01, University of Southern California - ISI, Feb. 1994.

1312. K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in software design. Technical Report Technical Report CS-2001-13, University of Virginia, 2001.

1313. K. Sullivan, P. Chalasani, and S. Jha. Software design decisions as real options. Technical Report Technical Report 97-14, University of Virginia, June SULLIV 1.PDF.

1314. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.

1315. P. Suppes, M. Cohen, R. Laddaga, J. Anliker, and R. Floyd. A procedural theory of eye movements in doing arithmetic. *Journal of Mathematical Psychology*, 27:341–369, 1983.

1316. D. C. Suresh, S. R. Mohanty, W. A. Najjar, L. N. Bhuyan, and F. Vahid. Loop level analysis of security and network applications. In *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-03)*, Feb. 2003.

1317. H.-M. Süß, K. Oberauer, W. W. Wittmann, O. Wilhelm, and R. Schulze. Working memory capacity and intelligence: An integrative approach based on brunswik symmetry. Technical report, Universität Mannheim, 1996.

1318. H. Sutter and A. Alexandrescu. *C++ Coding Standards: Rules, Guidelines, and Best Practices*. Addison Wesley, 2004.

1319. K. Svatko. Descriptive adjective ordering in English and Arabic. Quoted from by Celce-Murcia,[213] 1979.

1320. M. Swan and B. Smith. *Learner English*. Cambridge University Press, second edition, 2001.

1321. E. B. Swanson. IS "maintainability":Should it reduce the maintenance effort. *The DATA BASE for Advances in Information Systems*, 30(1):65–76, 1999.

1322. H. L. Swanson. What develops in working memory? A life span perspective. *Developmental Psychology*, 35(4):986–1000, 1999.

1323. D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965.

1324. P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 324–332, June 1998.

1325. J. Sweller, J. F. Mawer, and M. R. Ward. Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General*, 112(4):639–661, 1983.

1326. M. Taft. Lexical access via orthographic code: The basic orthographic syllabic structure (BOSS). *Journal of Verbal Learning and Verbal Behavior*, 18:21–39, 1979.

1327. M. Taft and K. I. Foster. Lexical storage and retrieval of prefixed words. *Journal of Verbal Learning and Verbal Behavior*, 14:638–647, 1975.

1328. M. Taft and K. I. Foster. Lexical storage and retrieval of polymorphemic and polysyllabic words. *Journal of Verbal Learning and Verbal Behavior*, 15:607–620, 1976.

1329. S. Talbott. *Managing Projects with Make*. O'Reilly & Associates, Inc, 1989.

1330. L. H. Tan and C. A. Perfetti. Phonological activation in visual identification of Chinese two-character words. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(2):382–393, 1999.

1331. J. W. Tanaka and M. Taylor. Object categories and expertise: Is the basic level in the eye of the beholder. *Cognitive Psychology*, 23:457–482, 1991.

1332. S. E. Taylor and J. D. Brown. Illusion and well-being: A social psychological perspective on mental health. *Psychological Bulletin*, 103(2):193–210, 1988.

1333. W. J. Teahan and J. G. Cleary. The entropy of English using PPM-based models. In *IEEE Data Compression Conference*, pages 53–62. IEEE Computer Society Press, 1996.

1334. B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology*, 79(1):142–155, 1994.

1335. D. Tennenhouse. It's time to get physical. In 20^{th} *IEEE Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

1336. K. Tentori, D. Osherson, L. Hasher, and C. May. Wisdom and ageing: Irrational preferences in college students but not older adults. *Cognition*, 81(3):B87–B99, 2001.

1337. P. E. Tetlock. Accountability: The neglected social context of judgment and choice. *Research in Organizational Behavior*, 7:297–332, 1985.

1338. P. E. Tetlock. An alternative metaphor in the study of judgment and choice: People as politicians. *Theory and Psychology*, 1(4):451–475, 1991.

1339. P. E. Tetlock, O. V. Kristel, S. B. Elson, M. C. Green, and J. S. Lerner. The psychology of the unthinkable: Taboo trade-offs, forbidden base rates, and heretical counterfactuals. *Journal of Personality and Social Psychology*, 78:853–870, 2000.

1340. Texas Instruments. *TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler Users' Guide*. Texas Instruments, spnu022c edition, Apr. 1996.

1341. Texas Instruments. *TMS320C3x User's Guide*. Texas Instruments, Inc, spru031e, revision l edition, July 1997.

1342. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, spru189f edition, Oct. 2000.

1343. Texas Instruments. *TMS320C6000 Programmer's Guide*. Texas Instruments, Inc, spru196d edition, Mar. 2000.

1344. T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland Publishing Company, 1978.

1345. The Santa Cruz Operation. *System V Application Binary Interface: MIPS RISC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, 3rd edition, Feb. 1996.

1346. The Santa Cruz Operation. *System V Application Binary Interface: SPARC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, third edition, 1996.

1347. The Santa Cruz Operation. *System V Application Binary Interface: Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, fourth edition, Mar. 1997.

1348. J. Thiyagalingam. *Alternative Array Storage Layout for Regular Scientific Programs*. PhD thesis, Imperial College, University of London, June 2005.

1349. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays, yet? *Concurrency and Computation: Practice & Experience*, 18(11):1509–1539, Sept. 2006.

1350. K. Thompson. Plan 9 C compilers. In *Plan 9 Programmer's Manual*. AT&T Bell Laboratories, 1995.

1351. P. Thompson. Margaret Thatcher: a new illusion. *Perception*, 9:483–484, 1980.

1352. M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

1353. M. A. Tinker. The relative legibility of the letters, the digits, and of certain mathematical signs. *Journal of Generative Psychology*, 1:472–494, 1928.

1354. H. Tomiyama and H. Yasuura. Optimal code placement of embedded software for instruction caches. In *European Design and Test Conference (ED&TC '96)*, pages 96–101. IEEE, Mar. 1996.

1355. P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari. Reverse engineering 4.7 million lines of code. *Software–Practice and Experience*, 30(2):129–150, Feb. 2000.

1356. Y. F. Tong, R. A. Rutenbar, and D. F. Nagle. Minimizing floating-point power dissipation via bit-width reduction. In *Proceedings of the 1998 International Symposium on Computer Architecture Power Driven Microarchitecture Workshop*, June 1998.

1357. T. Q. M. S. Tool. Botsch. freshmeat.net/projects/qmc, 2003.

1358. J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.

1359. J. T. Townsend. Alphabetic confusion: A test for individuals. *Perception & Psychophysics*, 9(6):449–454, 1971.

1360. J. T. Townsend. Theoretical analysis of an alphabetic confusion matrix. *Perception & Psychophysics*, 9(1A):40–50, 1971.

1361. R. Treiman and C. Barry. Dialect and authography: Some differences between American and British spellers. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 26:1423–1430, 2000.

1362. R. Treiman and B. Kessler. Context sensitivity in the spelling of English vowels. *Journal of Memory and Language*, 47:448–468, 2002.

1363. A. Treisman and J. Souther. Search asymmetry: A diagnostic for preattentive processing of separable features. *Journal of Experimental Psychology: General*, 114(3):285–310, 1985.

1364. A. M. Treisman and G. Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12:97–136, 1980.

1365. L. M. Trick and Z. W. Pylyshyn. What enumeration studies can show us about spatial attention: Evidence for limited capacity preattentive processing. *Journal of Experimental Psychology: Human Perception and Performance*, 19(2):331–351, 1993.

1366. L. M. Trick and Z. W. Pylyshyn. Why are small and large numbers enumerated differently? A limited-capacity preattentive stage in vision. *Psychological Review*, 101(1):80–102, 1994.

1367. C.-W. Tseng. Software support for improving locality in advanced scientific codes. Technical Report CS-TR-4168, University of Maryland, 2000.

1368. R. M. Tubbs, W. F. Messier Jr., and W. R. Knechel. Recency effects in the auditor's belief revision process. *The Accounting Review*, 65(2):452–460, Apr. 1990.

1369. J. Turley. Embedded processors. www.extremetech.com, Jan. 2002.

1370. R. T. Turley and J. M. Bieman. Competencies of exceptional and nonexceptional software engineers. *The Journal of Systems and Software*, 28(1):19–38, Jan. 1995.

1371. M. L. Turner and R. W. Engle. Is working memory capacity task dependent? *Journal of Memory and Language*, 28:127–154, 1989.

1372. A. Tversky. Elimination by aspects: A theory of choice. *Psychological Review*, 79(4):281–299, 1972.

1373. A. Tversky and D. Kahneman. Availability: A heuristic for judging frequency and probability. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 11, pages 163–178. Cambridge University Press, 1982.

1374. A. Tversky and D. Kahneman. Judgment under uncertainty: Heuristics and biases. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 1, pages 3–20. Cambridge University Press, 1982.

1375. A. Tversky and D. Kahneman. Judgments of and by representativeness. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 6, pages 84–98. Cambridge University Press, 1982.

1376. A. Tversky, S. Sattath, and P. Slovic. Contingent weighting in judgment and choice. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 28, pages 503–517. Cambridge University Press, 1999.

1377. A. Tversky and I. Simonson. Context-dependent preferences. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 29, pages 518–527. Cambridge University Press, 1999.

1378. R. D. Tweney, M. E. Doherty, W. J. Worner, D. P. Pliske, C. R. Mynatt, K. A. Gross, and D. L. Arkkelin. Strategies of rule discovery in an inference task. *Quarterly Journal of Experimental Psychology*, 32:109–123, 1980.

1379. F. Tydeman. Private email from Fred. www.tybor.com, 2002.

1380. F. Tydeman. Sample C99+FPCE tests. www.tybor.com, 2005.

1381. A. Tyler and V. Evans. Reconsidering prepositional polysemy networks: The case of over. *Language*, 77(4):724–765, 2001.

1382. A. Tyler and V. Evans. *The Semantics of English Prepositions*. Cambridge University Press, 2003.

1383. V. Tzerpos and R. C. Holt. A well-formedness theory of C source inclusion. In *Proceedings of the* 3^{rd} *International Symposium on Applied Corporate Computing (ISACC'95)*, pages 18–22, Oct. 1995.

1384. Ubicom. *IP2022 Internet Processor*. Ubicom, Inc, preliminary edition, July 2002.

1385. G.-R. Uh. *Effectively exploiting indirect jumps*. PhD thesis, Florida State University, 1997.

1386. G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao. Techniques for effectively exploiting a zero overhead loop buffer. In *Proceedings of the International Conference on Compiler Construction*, pages 157–172, Mar. 2000.

1387. G.-R. Uh and D. B. Whalley. Effectively exploiting indirect jumps. *Software–Practice and Experience*, 29(12):1061–1101, Oct. 1999.

1388. S. Unger. *Transforming Irreducible Regions of Control Flow into Reducible Regions by Optimized Node Splitting*. PhD thesis, Humboldt Universität zu Berlin, 1998.

1389. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.

1390. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.

1391. Unisys Corporation. *C Compiler Programming Reference Manual Volume 1: C Language and Library*. Unisys Corporation, 7831 0422'006, release level 8R1A edition, 2001.

1392. Unix System Laboratories, Inc. *Programmer's Guide: ANSI C Programming Support Tools*. Prentice Hall, Inc, 1990.

1393. Unix System Laboratories, Inc. *Unix System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. Prentice Hall, Inc, 1990.

1394. U.S. DoD. Memorandum on the use of the Ada programming language. Technical report, U.S. Department of Defence, Apr. 1997.

1395. V. Živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSP-STONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.

1396. T. Valentine, T. Brennen, and S. Breédart. *The Cognitive Psychology of Proper Names*. Routledge, 1996.

1397. M. L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, Oct. 2002.

1398. O. Van den Bergh, S. Vrana, and P. Eelen. Letters from the heart: Affective categorization of letter combinations in typists and nontypists. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 16(6):1153–1161, 1990.

1399. F. van der Velde, A. H. C. van der Heijden, and R. Schreuder. Context-dependent migrations in visual word perception. *Journal of Experimental Psychology: Human Perception and Performance*, 15(1):133–141, 1989.

1400. T. van Gelder. Penicillin for the mind? Reason, education and cognitive science. Technical Report Preprint No. 1/98, University of Melbourne Department of Philosophy, 1998.

1401. T. van Gelder and A. Bulka. Reason!: Improving informal reasoning skills. In *Proceedings of the Australian Computers in Education Conference*, 2000.

1402. W. J. B. van Heuven, T. Dijkstra, and J. Grainger. Orthographic neighborhood effects in bilingual word recognition. *Journal of Memory and Language*, 39:458–483, 1998.

1403. G. C. Van Orden. A ROWS is a ROSE: Spelling, sound, and reading. *Memory & Cognition*, 15(3):181–198, 1987.

1404. I. van Rooij. Notation specificity in numerical cognition: A critique of Noël and Seron's (1992) reappraisal of the Gonzalez and Kolers (1982) study. 2000.

1405. C. Van Rooy, C. Stough, A. Pipingas, C. Hocking, and R. B. Silberstein. Spatial working memory and intelligence Biological correlates. *Intelligence*, 29:275–292, 2001.

1406. R. J. J. H. van Son and L. C. W. Pols. An acoustic model of communicative efficiency in consonants and vowels taking into account context distinctiveness. In *Proceedings of ICPhS (International Congress of Phonetic Sciences)*, pages 2141–2144, Aug. 2003.

1407. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. *Algol 68*. Springer–Verlag, 1976.

1408. S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

1409. H. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. The MIT Press, 1990.

1410. J. J. Vannest. *Morphological Effects in Visual Word Processing: Their Timecourse and Consequences for Lexical Architecture*. PhD thesis, The Ohio State University, 2001.

1411. F. J. Varela, E. Thompson, and E. Rosch. *The Embodied Mind: Cognitive Science and Human Experience*. The MIT Press, 1999.

1412. A. G. Vartabedian. The effects of letter size, case, and generation method on CRT display search time. *Human Factors*, 13(4):363–368, 1971.

1413. R. L. Velduizen. C++ templates as partial evaluation. Technical Report TR519, Indiana University, July 2000.

1414. R. L. Venezky. From Webster to Rice to Roosevelt: The formative years for spelling instruction and spelling reform in the U.S.A. In U. Frith, editor, *Cognitive Processes in Spelling*, chapter 1, pages 9–30. Academic Press, 1980.

1415. R. L. Venezky. *The American Way of Spelling*. Guilford Press, 1999.

1416. C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation A study in C. In *Proceedings of the 1996 International Conference on Compiler Construction*, pages 74–89. Springer-Verlag, Apr. 1996.

1417. K. L. Verco and M. J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of First Australian Conference on Computer Science Education*, 1996.

1418. B. Verdonk, A. Cuyt, and D. Verschaeren. A precision and range independent tool for testing floating-point arithmetic I: basic operations, square root and remainder. *ACM Transactions on Mathematical Software*, 27(1):92–118, 2001.

1419. P. Verghese and D. G. Pelli. The information capacity of visual attention. *Vision Research*, 32(5):983–995, 1992.

1420. P. Verhaeghen and J. Cerella. Ageing, executive control, and attention: a review of meta-analyses. *Neuroscience and Biobehavioral Reviews*, 26(7):849–857, 2002.

1421. P. Vickers. *CAITLIN: Implementation of a Musical Program Auralisation System to Study the Effects of Debugging Tasks as Performed by Novice Pascal Programmers*. PhD thesis, Loughborough University, 1999.

1422. J. Villarreal, R. Lysecky, S. Cotteral, and F. Vahid. A study of the loop behavior of embedded programs. Technical Report UCR-CSE-01-03, University of California, Riverside, Dec. 2001.

1423. G. Visaggio. Value-based decision model for renewal processes in software maintenance. Technical Report ISERN-99-06, Department of Informatics, University of Bari, Italy, 1999.

1424. E. Visser. Scannerless generalised-LR parsing. Technical Report Report P9707, University of Amsterdam, Aug. 1997.

1425. T. Vitale. An algorithm for high accuracy name pronunciation by parametric speech synthesizer. *Computational Linguistics*, 17(3):258–276, 1991.

1426. P. C. Vitz and B. S. Winkler. Predicting the judged "similarity of sound" of English words. *Journal of Verbal Learning and Verbal Behavior*, 12:373–388, 1973.

1427. J. Voas, L. Morell, and K. Miller. Using dynamic sensitivity analysis to assess testability. www.cigital.com, 1991.

1428. K. von Fintel and L. Matthewson. Universals in semantics. *The Linguistic Review*, ???(???):???, Oct. 2007.

1429. A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements in an industrial environment. In *Proceedings Second Workshop on Program Comprehension*, pages 78–86, July 1993.

1430. A. von Mayrhauser, A. M. Vans, and S. Lang. Program comprehension and enhancement of software. In *Proceedings IFIP World Computing Congress - Information Technology and Knowledge Engineering*, 1998.

1431. W. Vonk, R. Radach, and H. van Rijn. Eye guidance and the saliency of word beginnings in reading text. In A. Kennedy, R. Radach, D. Heller, and J. Pynte, editors, *Reading as a Perceptual Process*, chapter 11, pages 269–299. North-Holland, 2000.

1432. J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001)*, pages 155–164, 2001.

1433. T. A. Wagner and S. L. Graham. General incremental lexical analysis. http://harmonia.cs.berkeley.edu/harmonia/papers/twagner-lexing.pdf, 1997.

1434. T. A. Wagner and S. L. Graham. Modeling user-provided whitespace and comments in an incremental SDE. *Software–Practice and Experience*, ???(1):???, 1998.

1435. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, Dec. 1993.

1436. W. M. Waite. The cost of lexical analysis. *Software–Practice and Experience*, 16(5):473–488, 1986.

1437. D. W. Wall. Global register allocation at link time. Technical Report 86/3, Western Research Lab, Oct. 1986.

1438. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc, 3rd edition, 2000.

1439. S. R. Walli. The myth of application source-code conformance. *Standard View*, 4(2):94–99, June 1996.

1440. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, Dec. 2000.

1441. C. Ware. *Information Visualization Perception for Design*. Morgan Kaufmann Publishers, 2000.

1442. W. H. Ware, S. N. Alexander, P. Armer, M. M. Astrahan, T. Biers, H. H. Goode, H. D. Huskey, and M. Rebinoff. Soviet computer technology – 1959. *Communications of the ACM*, 3(3):131–166, 1960.

1443. J. H. S. Warren. *Hacker's Delight*. Addison–Wesley, 4th edition, 2003.

1444. P. C. Wason. On the failure to eliminate hypothesis in a conceptual task. *Quarterly Journal of Experimental Psychology*, XII:129–140, 1960.

1445. P. C. Wason. Reasoning. In B. M. Foss, editor, *New Horizons in Psychology 1*, chapter 6, pages 135–151. Penguin Books, 1966.

1446. Watcom. *Watcom C Language Reference*. Sybase, Inc, 11.0c edition, 2000.

1447. A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report NIST Special Publication 500-235, National Institute of Standards and Technology (NIST), Sept. 1996.

1448. J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, Mar. 1996.

1449. T. P. Way. *Procedure restructuring for ambitious optimization*. PhD thesis, University of Delaware, 2002.

1450. D. L. Weaver and T. Germond. *The SPARC Architecture Manual*. Prentice Hall, Inc, ninth edition, 2000.

1451. B. S. Weekes. Differential effects of number of letters on word and nonword naming latency. *Quarterly Journal of Experimental Psychology*, 50A(2):439–456, 1997.

1452. S. Wells-Jensen. *Cognitive Correlates of Linguistic Complexity: A Cross-Linguistic Comparison of Errors in Speech*. PhD thesis, State University of New York at Buffalo, July 1999.

1453. J. Welsh and A. Hay. *A Model Implementation of Standard Pascal*. Prentice-Hall, Inc, 1986.

1454. J. W. Whalen. *The Influence of Semantic Number representations on Arithmetic Fact Retrieval*. PhD thesis, The John Hopkins University, July 2000.

1455. R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 45–54, Sept. 2003.

1456. T. White House. Guidelines and discount rates for benefit-cost analysis of federal programs. Technical Report OMB Circular A-94, US Government, 1992.

1457. B. A. Wichmann and M. Davies. Experience with a compiler testing tool. Technical Report NPL Report DITC 138/89, National Physical Laboratory, England, 1989.

1458. W. A. Wickelgren. Size of rehearsal group and short-term memory. *Journal of Experimental Psychology*, 68(4):413–419, 1964.

1459. W. A. Wickelgren. Short-term memory for repeated and non-repeated items. *Quarterly Journal of Experimental Psychology*, 17:14–25, 1965.

1460. C. A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Proceedings of the first International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177–184, Mar. 1982.

1461. A. Wierzbicka. *Semantics: Primes and Universals*. Oxford University Press, 1996.

1462. T. A. Wiggerts. Using clustering algorithms in legacy systems remodularizations. In 4^{th} *Working Conference on Reverse Engineering (WCRE '97)*, pages 33–43. IEEE, Oct. 1997.

1463. J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In 10^{th} *Network and Distributed System Security Symposium (NDSS'03)*, pages 149–162, Feb. 2003.

1464. E. Wiles. Economic models of software reuse: A survey, comparison and partial validation. Technical Report UWA-DCS-99-032, Department of Computer Science, University of Wales, Aberystwyth, 1999.

1465. J. Wiley. Expertise as mental set: The effects of domain knowledge in creative problem solving. *Memory & Cognition*, 26(4):716–730, 1998.

1466. J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover, 1994.

1467. E. D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.

1468. E. D. Willink and V. B. Muchnick. Preprocessing C++: Substitution and composition. In *Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems*, June 1999.

1469. T. C. Willoughby. Are programmers paranoid? In *Proceedings of the Tenth Annual SIGCPR Conference*, pages 47–54, June 1972.

1470. L. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992.

1471. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. G. Baker, editor, *Proceedings 1995 International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, Berlin, Germany, Sept. 1995.

1472. T. D. Wilson, S. Lindsey, and T. Y. Schooler. A model of dual attitudes. *Psychological Review*, 107(1):101–126, 2000.

1473. A. M. Wing and A. D. Baddeley. Spelling errors in handwriting: A corpus and distributional analysis. In U. Frith, editor, *Cognitive Processes in Spelling*, chapter 12, pages 251–285. Academic Press, 1980.

1474. D. S. Wise and J. D. Frens. Morton-order matrices deserve compilers' support. Technical Report Technical Report 533, Indiana University, Nov. 1999.

1475. J. C. Wise, D. L. Hannaman, P. Kozumplik, E. Franke, and B. L. Leaver. Methods to improve cultural communication skills in special operations forces. Technical Report ARI Contract Report 98-06, United States Army Research Institute for the Behavioral and Social Sciences, July 1998.

1476. E. Wisniewski and B. C. Love. Relations versus properties in conceptual combination. *Journal of Memory and Language*, 38:177–202, 1998.

1477. E. J. Wisniewski and D. Gentner. On the combinatorial semantics of noun pairs: Minor and major adjustments to meaning. In G. B. Simpson, editor, *Understanding word and sentence*, pages 241–284. Amsterdam: North Holland, 1991.

1478. J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Nov. 1996.

1479. WL | Delft Hydraulics. Programmer's guide C programming rules. Technical Report OMS report number 2001-02, WL | Delft Hydraulics, Nov. 2001.

1480. M. Wolfe. How compilers and tools differ for embedded systems. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2005*, pages 1–4. ACM, Sept. 2005.

1481. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the* 22^{nd} *Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

1482. S. Woods. *A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering*. PhD thesis, University of Waterloo, 1996.

1483. S. Woods and Q. Yang. Approaching the program understanding problem: Analysis and A heuristic solution. In *Proceedings of the* 18^{th} *International Conference on Software Engineering (ICSE '96)*, Mar. 1996.

1484. L. Wu, C. Weaver, and T. Austin. CryptoManiac: A fast flexible architecture for secure communication. In 28^{th} *Annual International Symposium on Computer Architecture (ISCA-2001)*, pages 110–119, June 2001.

1485. L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W.-M. W. Hwu. A new framework for debugging globally optimized code. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, 1999.

1486. Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, Nov. 2002.

1487. X/Open Company Ltd. *Go Solo –How to implement and Go Solo with the Single UNIX Specification*. Prentice Hall, Inc, 1995.

1488. Z. Xu, T. Reps, and B. P. Miller. Typestate checking of machine code. In *European Symposium On Programming 2001*, 2001.

1489. J. Yang and R. Gupta. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems*, 2(3):1–27, 2002.

1490. M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):667–697, 2002.

1491. E. J. Yannakoudakis and D. Fawthrop. The rules of spelling errors. *Information Processing & Management*, 19(2):87–99, 1983.

1492. J. J. Yi and D. J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *International Conference on Computer Design (ICCD'02)*, pages 462–467, Sept. 2002.

1493. P. N. Yianilos and K. G. Kanzelberger. The LIKEIT intelligent string comparison facility. Technical report, NEC Research Institute, May 1997.

1494. H. Yokoo. Overflow/underflow-free floating-point number representations with self-delimiting variable-length exponent field. *IEEE Transactions on Computers*, 41(8):1033–1039, Aug. 1992.

1495. C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. *SIGPLAN Notices*, 32(5):183–193, May 1997.

1496. W. D. Young. *A verified code generator for a subset of Gypsy*. PhD thesis, The University of Texas at Austin, Dec. 1988.

1497. W. D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, Apr.-June 1998.

1498. R. Yung. Evaluation of a commercial microprocessor. Technical Report SMLI TR-98-65, Sun Microsystems, 1998.

1499. Z-World. *Dynamic C User's Manual*. Z-World, Inc, Davis, CA, USA, 019-0071.020218-p edition, 1999.

1500. E. M. Zamora, J. J. Pollock, and A. Zamora. The use of trigram analysis for spelling error detection. *Information Processing & Management*, 17(6):305–316, 1981.

1501. M. J. Zastre. Compacting object code via parameterized procedure abstraction. Thesis (m.s.), University of Victoria, 1993.

1502. N. J. Zbrodoff. Why is 9 + 7 harder than 2 + 3? Strength and interference as explanations of the problem-size effect. *Memory & Cognition*, 23(6):689–700, 1995.

1503. N. J. Zbrodoff and G. D. Logan. On the relation between production and verification tasks in the psychology of simple arithmetic. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 16(1):83–97, 1990.

1504. S. F. Zeigler. Comparing development costs of C and Ada. Technical report, Rational Software Corporation, Mar. 1995.

1505. M. V. Zelkowitz and D. Wallace. Experimental models for validating computer technology. *IEEE Computer*, 31(5):23–31, May 1998.

1506. J. D. Zevin and M. S. Seidenberg. Age of acquisition effects in word reading and other tasks. *Journal of Memory and Language*, 47:1–29, 2002.

1507. J. Zhang and H. J. Hamilton. Learning English syllabification rules. In R. E. Mercer and E. Neufeld, editors, *Advances in Artificial Intelligence:* 12^{th} *Biennial Conference of the Canadian AI Society for the Computational Studies of Intelligence (AI'98) Proceedings*, pages 246–258, 1998.

1508. J. Zhang and H. Wang. The effect of external representations on numeric tasks. *Quarterly Journal of Experimental Psychology*, 58(5):817–838, Oct. 2005.

1509. S. Zhang and B. G. Ryder. Complexity of single level function pointer aliasing analysis. Technical Report LCSR-TR-233, Rutgers University, Aug. 1994.

1510. X.-X. S. Zhang. *Practical pointer aliasing analysis*. PhD thesis, New Brunswick Rutgers, The State University of New Jersey, Oct. 1998.

1511. Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In R. N. Horspool, editor, *Compiler Construction* 11^{th} *International Conference, CC 2002*, pages 14–28. Springer-Verlag, Apr. 2002.

1512. J. C. Ziegler, C. Perry, A. M. Jacobs, and M. Braun. Identical words are read differently in different languages. *Psychological Science*, 12(5):379–384, Sept. 2001.

1513. J. C. Ziegler, A. Rey, and A. M. Jacobs. Simulating individual word identification thresholds and errors in the fragmentation task. *Memory & Cognition*, 26(3):490–501, 1998.

1514. G. K. Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison–Wesley, 1949.

1515. F. Zlotnick. *The POSIX.1 Standard: A Programmer's Guide*. The Benjamin/Cummings Publishing Company, 1991.

1516. J. Zobel. Reliable research: Towards experimental standards for computer science. In *Proceedings of the* 21^{st} *Australian Computer Science Conference*, pages 217–229, Feb. 1998.

1517. S. Zucker and K. Karhi. *System V Application Binary Interface: PowerPC processor Supplement*. SunSoft, Mountain View, CA, USA, 802-3334-10 edition, Sept. 1995.