9

Tipo Abstracto de Dato (TAD)

Contenido

9.1 Introducción 234
9.2 Capas de abstracción 234
9.3 Tipos de datos 235
9.4 Resumen 248
9.5 Contenido de la página Web de apoyo 248

Objetivos del capítulo

- Analizar y descubrir la existencia de diferentes capas de abstracción.
- Comprender el concepto de "tipo de dato abstracto" (TAD)
- Analizar los tipos `FILE`, `FILE*` y la operaciones asociadas: `fopen`, `fread`, `fwrite`, etcétera.
- Lograr abstracción mediante el ocultamiento de la lógica algorítmica.
- Desarrollar el TAD `Fecha`, simple y como introducción al tema.
- Desarrollar el TAD `XFile` para encapsular la técnica de "baja lógica" sobre los registros de un archivo.

Competencias específicas

- Representar y aplicar los tipos de datos abstractos por medio de un lenguaje de programación C.

9.1 Introducción

Generalmente, uno de los primeros programas que desarrollamos cuando comenzamos a estudiar programación consiste en pedirle al usuario que ingrese dos valores numéricos, sumarlos y mostrar el resultado en la consola.

Este tipo de programas no reviste ningún tipo de complejidad porque el lenguaje de programación provee resuelta la funcionalidad necesaria para mostrar y leer datos a través de la consola y, además, para realizar operaciones aritméticas entre valores numéricos.

Los programadores que recién comienzan no se preocupan por lo difícil que le pueda resultar a la computadora la acción de sumar dos números enteros. Simplemente, los suman con el operador + y se abstraen del problema.

Si pudiéramos hacer extensible la naturalidad con la que, por ejemplo, sumamos valores numéricos a cualquier otro tipo de operación sobre cualquier otro tipo o estructura de datos, sin duda, estaremos dando un gran paso hacia el encapsulamiento y la abstracción.

Los tipos de dato abstractos, dentro de los límites de la programación estructurada, permiten desarrollar y extender al máximo los conceptos de encapsulamiento y abstracción. De esta manera, se facilita la manera de ocultar la lógica de las operaciones que son propias de un tipo o estructura para que el programador pueda abstraerse de estos temas y concentrarse en su verdadero problema: el desarrollo de la aplicación.

9.2 Capas de abstracción

Quizás sin darnos cuenta, hemos desarrollado ejemplos y programas sin tener que preocuparnos por cuestiones tales como mostrar un mensaje en la consola, leer un valor por teclado, grabar un registro en un archivo, etc. Para nosotros (como programadores) todos estos temas están resueltos porque otros programadores desarrollaron, entre otras, las funciones `printf`, `scanf`, `fread`, `fwrite`, etc, y las incluyeron como parte de la biblioteca estándar del lenguaje de programación.

A su vez, quienes programaron las funciones de biblioteca, seguramente, habrán encontrado muchas cuestiones resueltas provistas como parte de los servicios del sistema operativo (*system call*) que, por cierto, fue desarrollado por otros programadores.

Es decir, cuando programamos aplicaciones lo hacemos montados sobre una capa de abstracción que nos permite obviar cuestiones de plataforma que, en la mayoría de los casos, resultan totalmente ajenas al contexto de nuestro negocio.

Por ejemplo, si desarrollamos una aplicación para liquidar los sueldos de los empleados de una compañía, nuestro contexto (negocio) estará relacionado con “empleados”, “categorías”, “horas trabajadas”, etc. No tenemos que preocupamos por programar funciones que muestren mensajes en la consola o funciones que lean o escriban datos en archivos porque, para nosotros (programadores de aplicaciones), estos temas están (o deberían estar) resueltos.

Así, podemos identificar varios niveles o capas de abstracción donde cada capa resuelve o abstrae de cierto tipo de problemas a la capa superior y se apoya en la abstracción que le provee la capa inferior.

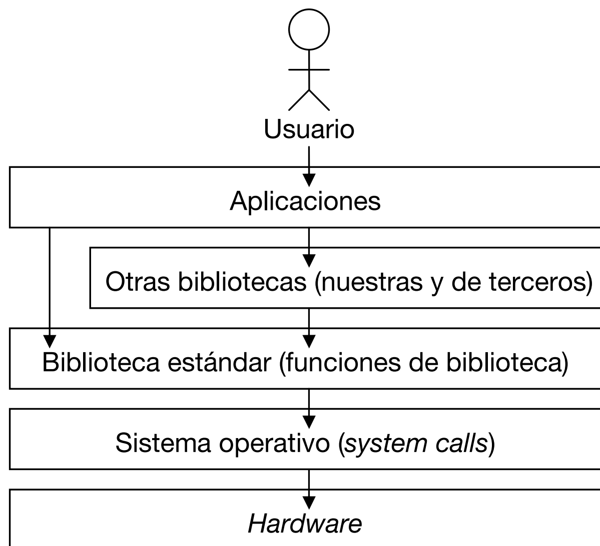

Fig. 9.1 Capas de abstracción.

El gráfico muestra al usuario final interactuando con una aplicación que fue programada invocando a funciones de la biblioteca estándar de C y otras desarrolladas por nosotros mismos o por terceras partes. Estas últimas, seguramente, fueron desarrolladas invocando a las funciones estándar de C. Las funciones de la biblioteca estándar invocan a los servicios provistos por el sistema operativo que, finalmente, interactúan con el *hardware* o plataforma física.

9.3 Tipos de datos

Como estudiamos en el Capítulo 1, el concepto de "tipo de datos" agrupa dos conjuntos:

- Un conjunto *V* de posibles valores que, por sus características o naturaleza, podemos identificar como "del mismo tipo".
- Un conjunto *T* de operaciones aplicables sobre los elementos del conjunto anterior.

Por ejemplo, analicemos el tipo de datos *Entero*:

Entero = {*V*, *T*}

V = { *x*/*x* es un valor numérico entero positivo, entero negativo o cero }

T = { *x*/*x* es un operador aritmético o un operador relacional }

Tal como está planteado, la definición teórica del tipo de datos *Entero* admite valores que podrían ir desde –*infinito* hasta +*infinito*. Sin embargo, en la práctica, los datos se almacenan en la memoria principal de la computadora, cuya capacidad de almacenamiento es finita. Esto hace que la diversidad de valores que pueda contener el conjunto *V* dependa de la cantidad de memoria que se designe para guardar a cada valor.

Concretamente, en C, el tipo de datos `int` es una implementación del tipo de datos *Entero* que admite valores dentro del siguiente intervalo: [-32768, +32767] (suponiendo que se utilizan 2 *bytes* para su representación). Y el tipo `long` también es una imple-

mentación del tipo *Entero* que admite los valores del intervalo: [-2^{15},$+2^{15}-1$] (suponiendo que se utilizan 4 *bytes* para su representación).

Volviendo a la idea de “abstracción”, el concepto de “tipo de datos” abstrae del problema de operar con valores de ese tipo. Es decir, leer dos valores numéricos enteros, sumarlos y mostrar el resultado por consola es una tarea trivial gracias a que el lenguaje de programación provee el tipo de datos `int` y las funciones `printf` y `scanf`.

9.3.1 Tipo Abstracto de Dato (TAD)

El concepto de TAD es una extensión del concepto de “tipo de datos” que nos permite diseñar nuestros propios tipos para encapsular lógica algorítmica y proveer abstracción a las capas de *software* de más alto nivel.

Sin ir más lejos, el tipo `FILE` y las operaciones (o funciones) `fopen`, `fclose`, `fread`, `fwrite`, etc., componen un TAD que encapsula la problemática relacionada con el manejo de archivos.

Probablemente, el lector no se haya percatado de que `FILE` no es un tipo de datos primitivo sino que es una estructura definida en el archivo `stdio.h`. Veamos:

```
typedef struct _iobuf
{
    char* _ptr;
    int _cnt;
    char* _base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char* _tmpfname;
}FILE;
```

Por suerte, no hemos tenido que estudiar el significado de los campos de esta estructura y tampoco tendremos que hacerlo, simplemente, porque son parte de la implementación del TAD `FILE`.

9.3.2 Interfaz e implementación de un TAD

Según el gráfico de capas de abstracción que analizamos más arriba, nosotros (como programadores) somos usuarios de las funciones que provee la biblioteca estándar de C. Utilizamos el tipo `FILE` para abrir, leer, escribir y cerrar archivos y no fue necesario conocer su estructura interna porque contamos con un conjunto de funciones que, apoyándose en una variable de este tipo, resuelven el problema y nos proveen abstracción.

El concepto de “interfaz” e “implementación” se presenta en la vida cotidiana. Por ejemplo, no es necesario graduarse de Ingeniero Electrónico para encender un televisor, sintonizar un canal y luego apagarlo. Cualquiera de nosotros lo puede hacer gracias a que el televisor tiene una interfaz que se compone (seguramente) de un botón de encendido, uno para avanzar y retroceder el canal, uno para ajustar el volumen, etcétera.

Nosotros somos “usuarios del televisor” y como tales sabemos que contamos con los servicios provistos por su interfaz. Luego, si queremos cambiar de canal presionamos el botón correspondiente, pero lo que suceda del botón hacia adentro del aparato no es nuestro problema. Es parte de la implementación de la función “cambiar canal”.

Supongamos que nuestro televisor es analógico y no funciona con las nuevas señales digitales, razón por la cual decidimos llevarlo a un técnico para que lo adapte. Lo que

el técnico haga dentro del televisor no es de nuestra incumbencia, siempre y cuando, al regresar a nuestra casa con el aparato, todo siga funcionando correctamente y podamos sintonizar también las señales digitales.

El técnico cambió la implementación de la funcionalidad del televisor para adaptarla a las nuevas tecnologías. Sin embargo, la interfaz continúa siendo la misma.

9.3.3 El TAD `Fecha`

En el Capítulo 7, definimos el TAD `Fecha` compuesto por el tipo `Fecha` y las funciones `crearFecha`, `obtenerDia`, `obtenerMes`, `obtenerAnio`, `compararFechas` y `toString`.

En aquel momento, optamos por implementar la fecha como un número entero de 8 dígitos con formato *aaaaddmm*. Así, definimos el tipo de datos `Fecha` de la siguiente manera:

```
typedef long Fecha;
```

y para probar su funcionamiento desarrollamos el siguiente programa:

`testFecha.c`

```
#include "fecha.h"
#include <stdio.h>

int main()
{
   // "creo" las fechas 2/10/1970 y 3/8/2010
   Fecha f1 = crearFecha(2,10,1970);
   Fecha f2 = crearFecha(3,8,2010);

   // obtengo sus representaciones con formato "dd/mm/aaaa"
   char* sF1 = toString(f1);
   char* sF2 = toString(f2);

   // las comparo para ver cual es posterior
   if( compararFechas(f1,f2)<0 )
   {
      printf("%s es posterior a %s\n",sF2,sF1);
   }
   else
   {
      printf("%s es posterior a %s\n",sF1,sF2);
   }

   return 0;
}
```

Veamos que sucede si decidimos cambiar la implementación del tipo de datos y reemplazar el entero de 8 dígitos por una estructura con los campos `dia`, `mes` y `anio`.

```
typedef struct Fecha
{
   int dia;
   int mes;
   int anio;
}Fecha;
```

Obviamente, ahora tendremos que modificar todas las funciones:

fechaImle2.c

```
// :
#include "fecha.h"

Fecha crearFecha(int d,int m ,int a)
{
   Fecha f;
   f.dia=f;
   f.mes=m;
   f.anio=a;

   return f;
}

int obtenerDia(Fecha f)
{
   return f.dia;
}

int obtenerMes(Fecha f)
{
   return f.mes;
}

int obtenerAnio(Fecha f)
{
   return f.anio;
}

int compararFechas(Fecha f1,Fecha f2)
{
   int difD=f2.dia-f1.dia;
   int difM=f2.mes-f1.mes;
   int difA=f2.anio-f1.anio;

   return difA!=0?difA:difM!=0?difM:difD;
}

char* toString(Fecha f)
{
   char* s=(char*)malloc(11);
   sprintf(s, "%02d/%02d/%04d",f.dia,f.mes,f.anio);

   return s;
}
```

Notemos que al cambiar la implementación no modificamos el prototipo de ninguna de las funciones definidas en `fecha.h`. Por esto, la interfaz del TAD `Fecha` continúa siendo la misma y, en consecuencia, el cambio no genera ningún impacto en los programas que utilizan este tipo de datos.

Le sugiero al lector que retroceda unas líneas y vuelva a analizar el código de `testFecha.c`. Podrá verificar que ese programa funciona perfectamente con la nueva implementación de `Fecha` ya que el nombre del tipo de datos y los prototipos de sus funciones asociadas no sufrieron ninguna modificación.

9.3.4 El TAD `XFile` (implementación de bajas lógicas)

En el capítulo anterior, analizamos dos estrategias de implementación de bajas lógicas sobre archivos de registros. Luego de este análisis pudimos determinar que la mejor solución consiste en utilizar un archivo auxiliar para almacenar los números de los registros que se van dando de baja.

Aquí profundizaremos esta estrategia y desarrollaremos un TAD para encapsular una implementación que, luego, nos posibilite eliminar registros de archivos secuenciales como si esto físicamente fuera posible.

El TAD `FILE` nos abstrae del problema de operar con archivos. La idea ahora es desarrollar un nuevo TAD para proveer al usuario (que será otro programador) un conjunto de funciones de manejo de archivos de registros con las cuales pueda leer, escribir y, sobre todo, eliminar registros. Veamos un ejemplo:

```
// abro el archivo
XFile f = xopen("EMP.dat","a+b",sizeof(Emp));

// elimino el registro numero 3
xdelete(&f,3);
```

En este ejemplo, abrimos el archivo `EMP.dat` y eliminamos el registro ubicado en la posición número 3. Tan simple como eso ya que las funciones `xopen` y `xdelete` hacen transparente para el usuario el hecho de que, por atrás, existe un archivo auxiliar.

9.3.4.1 Análisis de la estrategia

Como comentamos más arriba, la estrategia que utilizaremos para implementar bajas lógicas sobre cualquier archivo de registros consiste en mantener un archivo auxiliar para almacenar los números o posiciones de los registros eliminados.

Recordemos el ejemplo desarrollado en el capítulo anterior.

En principio, el archivo `ARCHIVO.dat` tiene todos sus registros, razón por la cual las posiciones físicas y lógicas de cada uno coinciden.

`ARCHIVO.dat`

	Reg0	Reg1	Reg2	Reg3	Reg4	Reg5	Reg6	Reg7	Reg8	Reg9
lógica	0	1	2	3	4	5	6	7	8	9
física	0	1	2	3	4	5	6	7	8	9

Luego, para dar de baja (por ejemplo) al registro ubicado en la posición número 3 grabaremos este valor en un nuevo archivo que, por ahora, llamaremos `BAJAS.dat`

`BAJAS.dat`

3
0

Si bien `ARCHIVO.dat` mantiene sus 10 registros físicos, desde el punto de vista lógico debemos considerar que tiene un registro menos. Esto hará que todos los registros que se ubican a partir de la posición 4 (inclusive) se desplacen una posición hacia adelante.

`ARCHIVO.dat`

	Reg0	Reg1	Reg2	~~Reg3~~	Reg4	Reg5	Reg6	Reg7	Reg8	Reg9
lógica	0	1	2		3	4	5	6	7	8
física	0	1	2	3	4	5	6	7	8	9

Si damos de baja el registro número 0 todos los registros ubicados en las posiciones posteriores se desplazarán un lugar.

`BAJAS.dat`

3	0
0	1

`ARCHIVO.dat`

	~~Reg0~~	Reg1	Reg2	~~Reg3~~	Reg4	Reg5	Reg6	Reg7	Reg8	Reg9
lógica		0	1		2	3	4	5	6	7
física	0	1	2	3	4	5	6	7	8	9

Por último, demos de baja al registro número 4.

`BAJAS.dat`

3	0	4
0	1	2

`ARCHIVO.dat`

	~~Reg0~~	Reg1	Reg2	~~Reg3~~	Reg4	Reg5	~~Reg6~~	Reg7	Reg8	Reg9
lógica		0	1		2	3		4	5	6
física	0	1	2	3	4	5	6	7	8	9

Como podemos ver, a medida que eliminamos registros se va incrementando el desfasaje entre las posiciones físicas y las posiciones lógicas de los registros que quedan.

Esta estrategia requiere que cualquier operación que vayamos a programar acceda, al menos, a las siguientes estructuras:

- El archivo original.
- El archivo de bajas.
- Un *array* (y su correspondiente longitud) para mantener en memoria las posiciones de los registros eliminados.
- Además, como vamos a trabajar sobre archivos de registros, las operaciones probablemente necesiten conocer la longitud de su tipo de datos. Entonces, para simplificar la tarea vamos a unificar todas estas variables en un único tipo que llamaremos `XFile`.

`xfile.h`

```
typedef struct XFile
{
   FILE* arch;       // archivo original
   int recSize;      // longitud de los registros del archivo original
   FILE* archBajas; // archivo de bajas
   int* bajas;       // array de bajas para subir el archivo
   int lenBajas;     // longitud del array de bajas
   int actual;       // numero de registro actualmente apuntado en arch
}XFile;
// :
```

La estructura `XFile` permite guardar los punteros al archivo original (`arch`) y al archivo de bajas (`archBajas`). También tiene un *array* (`bajas`) que nos permitirá mantener en memoria la información de los registros eliminados, su longitud (`lenBajas`) y una referencia al registro (lógico) que actualmente está siendo apuntado (`actual`).

Notemos que el *array* `bajas` está definido como `int*`. Su dimensión la asignaremos en el momento de abrir el archivo `arch`, en función de la cantidad de registros que contenga.

Veamos las operaciones (o funciones) aplicables sobre el tipo `XFile`:

```
// :

// abrir el archivo
XFile xopen(char*,char*,int);

// cerrar el archivo
void xclose(XFile*);

// leer un registro
void xread(XFile*, void*);

// escribir un registro
void xwrite(XFile*, void*);

// determinar si llego el fin del archivo
int xeof(XFile*);

// eliminar un registro
void xdelete(XFile*,int);

// mover el indicador de posicion del archivo
void xseek(XFile*,int);

// obtener el tamanio del archivo
long xfileSize(XFile*, int raw);

// obtener la cantidad de registros
int xrecCount(XFile*,int raw);

// posiciona el puntero en el registro logico numero 0
void xreset(XFile*);

// retorna el numero de registro logico que actualment esta siendo apuntado
int xcurrent(XFile*);
```

Analizaremos cada una, comenzando por las más simples.

9.3.4.2 Función `xopen`

```
XFile xopen(char* filename, char* mode, int recSize)
```

Esta función recibe el nombre del archivo (`filename`), el modo de apertura (`mode`) y el tamaño (en *bytes*) que ocupa cada uno de sus registros (`recSize`). Si invocamos a `xopen` para abrir el archivo `EMP.dat` (que contiene registros de tipo `Emp`) deberíamos hacerlo de la siguiente manera:

```
// abro el archivo
XFile f = xopen("EMP.dat","a+b",sizeof(Emp));
```

Como vemos, la función retorna un valor de tipo `XFile`. Este valor debemos mantenerlo en memoria porque es un argumento requerido por todas las funciones “x” provistas con el TAD.

Veamos la codificación de la función:
xfile.c

```
// :
XFile xopen(char* filename, char* mode, int recSize)
{
   XFile f;

   // abro el archivo original
   f.arch=fopen(filename, mode);

   // abro el archivo de bajas (primero obtengo su nombre)
   char *filenameBajas=obtenerNombreBajas(filename);
   f.archBajas=fopen(filenameBajas, mode);

   // dimensiono el array de bajas y subo el archivo de bajas
   f.bajas=(int*)malloc(sizeof(int)*xrecCount(&f,1));
   f.lenBajas=subirBajas(f.archBajas, f.bajas);

   // guardo el tamanio de registro
   f.recSize=recSize;

   // posiciono el puntero en el primer registro del archivo
   xseek(&f, 0);
   return f;
}
// :
```

Las función `obtenerNombreBajas` retorna el nombre físico que, a partir del nombre del archivo original, asignaremos al archivo de bajas. Por otro lado, la función `subirBajas` recorre el archivo de bajas y sube cada uno de sus registros al *array* `f.bajas` retornando su longitud. Veamos la codificación de estas funciones:
xfile.c

```
// :
char* obtenerNombreBajas(char* filename)
{
   char* c=(char*) malloc(strlen(filename)+1+1);
   strcpy(c, filename);
   strcat(c, "_");

   return c;
}

int subirBajas(FILE* f, int a[])
{
   int r;
   int len=0;

   fread(&r, sizeof(int), 1, f);
   while( !feof(f) )
   {
      a[len++]=r;
      fread(&r, sizeof(int), 1, f);
   }

   return len;
}
// :
```

Notemos que estas dos funciones no están prototipadas en `xfile.h` porque **no forman parte de la interfaz** del TAD. Simplemente, son funciones internas que ayudan a estructurar mejor la lógica de la función `xopen`.

9.3.4.3 Función `xclose`

`void xclose(XFile* f)`

Aquí recibimos el `XFile` y tenemos que cerrar los dos archivos que este encapsula (el archivo original y el archivo de bajas). También debemos liberar la memoria que, dinámicamente, asignamos al *array* `bajas`.

xfile.c

```
// :
void xclose(XFile* f)
{
   fclose(f->archBajas);
   fclose(f->arch);
   free(f->bajas);
}
// :
```

9.3.4.4 Función `xdelete`

`void xdelete(XFile* f, int recNo)`

El objetivo aquí es “eliminar” el registro cuya posición lógica es `recNo`. Para esto, simplemente, tenemos que agregar este valor al final del archivo `archBajas` y agregarlo también al *array* `bajas` para tenerlo disponible en memoria.

xfile.c

```
// :
void xdelete(XFile* f, int recNo)
{
   // inserto en el array el numero de registro que se da de baja
   f->bajas[f->lenBajas++]=recNo;

   // grabo este numero al final del archivo de bajas
   fseek(f->archBajas, 0, SEEK_END);
   fwrite(&recNo, sizeof(int), 1, f->archBajas);
}
// :
```

9.3.4.5 Función `xseek`

Esta es la función clave para el éxito de la implementación del TAD. Veamos su cabecera:

`void xseek(XFile* f, int recNo)`

La función recibe el `XFile` y el número de registro lógico sobre el que nos debemos posicionar. Es muy importante entender la diferencia que existe entre una posición lógica y una posición física.

Volvamos a analizar el ejemplo estudiado más arriba donde, luego de haber eliminado los registros 3, 0 y 4, el archivo se veía así:

ARCHIVO.dat

	~~Reg0~~	Reg1	Reg2	~~Reg3~~	Reg4	Reg5	~~Reg6~~	Reg7	Reg8	Reg9
lógica		0	1		2	3		4	5	6
física	0	1	2	3	4	5	6	7	8	9

Para posicionarnos (por ejemplo) en el registro lógico número 3 (Reg5) debemos mover el puntero del archivo al registro físico número 5. Es decir, tenemos que hallar una relación que, en función de las bajas existentes, permita convertir una posición lógica en una posición física.

La idea es la siguiente: si cada vez que eliminamos un registro ubicado en una determinada posición, todos los registros subsiguientes "bajan", la operación inversa será recorrer el *array* `bajas` sumando 1 por cada registro eliminado cuya posición sea menor o igual a la posición que estamos procesando.

Así, desarrollaremos la función `regLogicoToRegFisico` de la siguiente manera:

xfile.c

```
// :
int regLogicoToRegFisico(XFile* f, int regLogico)
{
   int regFisico=regLogico;

   int i;
   for( i=f->lenBajas-1; i>=0; i-- )
   {
      if( f->bajas[i] <= regFisico )
      {
         regFisico++;
      }
   }

   return regFisico;
}

// :
```

Le sugiero al lector que realice la prueba de escritorio para esta función considerando los valores del ejemplo con el que estamos trabajando. Podrá verificar lo siguiente:

regLogicoToRegFisico(*x*)	**Valor de retorno**
regLogicoToRegFisico(0)	1
regLogicoToRegFisico(1)	2
regLogicoToRegFisico(2)	4
regLogicoToRegFisico(3)	5
regLogicoToRegFisico(4)	7
regLogicoToRegFisico(5)	8
regLogicoToRegFisico(6)	9

Ahora, la función `xseek` se puede resolver transformando la posición lógica en una posición física e invocando a `fseek` para posicionar el puntero en el archivo original.

xfile.c

```
// :
void xseek(XFile* f, int recNo)
{
   // convierto la posicion logica en una posicion fisica
   int p=regLogicoToRegFisico(f, recNo);

   // posiciono el registro fisico
   fseek(f->arch, p * f->recSize, SEEK_SET);

   // guardo el numero de registro actual (logico)
   f->actual=recNo;
}
// :
```

9.3.4.6 Funciones `xread` y `xwrite`

Estas funciones simplemente deben leer o escribir en el registro actual (previamente posicionado por `xseek`) y luego avanzar el puntero hacia el siguiente registro lógico del archivo preparándolo para la próxima lectura o escritura.

xfile.c

```
// :
void xread(XFile* f, void* reg)
{
   fread(reg, f->recSize, 1, f->arch);
   xseek(f, f->actual+1);
}

void xwrite(XFile* f, void* reg)
{
   fwrite(reg, f->recSize, 1, f->arch);
   xseek(f, f->actual+1);
}
// :
```

9.3.4.7 Función `xfileSize`

```
long xfileSize(XFile* f, int raw)
```

Esta función recibe el `XFile` y un *flag* (`raw`) que indica si queremos que devuelva el tamaño del archivo físico o el tamaño del archivo lógico (esto es: sin considerar los registros que fueron dados de baja).

xfile.c

```
// :
long xfileSize(XFile* f, int raw)
{
   // obtengo el tamanio del archivo fisico
   long actual=ftell(f->arch);
   fseek(f->arch, 0, SEEK_END);
   long ultimo=ftell(f->arch);
   fseek(f->arch, actual, SEEK_SET);
```

```
   // calculo la cantidad de bytes dados de baja
   long bytesBajas=f->lenBajas * f->recSize;

   // segun el flag retorno el size fisico y este menos las bajas
   return raw ? ultimo : ultimo-bytesBajas;
}
// :
```

9.3.4.8 Función `xrecCount`

`int xrecCount(XFile* f, int raw)`

Esta función es similar a la anterior, pero retorna la cantidad de registros que tiene el archivo. También recibe un *flag* que indica si hablamos del archivo físico o del archivo lógico.

xfile.c

```
// :
int xrecCount(XFile* f, int raw)
{
   return xfileSize(f, raw) / f->recSize;
}
// :
```

9.3.4.9 Función `xeof`

Esta función debe retornar *true* o *false* según se haya llegado o no al final del archivo. Esto lo podemos determinar en función de que la posición actual del `XFile` sea mayor que la cantidad de registros lógicos del archivo.

xfile.c

```
// :
int xeof(XFile* f)
{
   return f->actual > xrecCount(f, 0);
}
```

Por último, veamos el código de las funciones `xreset` y `xcurrent`. La primera posiciona el puntero del archivo para hacerlo apuntar al registro lógico número 0. La segunda retorna el número de registro que actualmente está siendo apuntado.

xfile.c

```
// :
void xreset(XFile* f)
{
   xseek(f,0);
}

long xcurrent(XFile* f)
{
   return f->actual;
}
```

Veamos ahora el código de un programa en el cual le mostramos al usuario todo el contenido del archivo `EMP.dat` y le pedimos que ingrese el número de registro que desea eliminar. Luego eliminamos el registro indicado invocando a la función `xdelete` y repetimos la operación hasta que el usuario ingrese un valor negativo.

testXFile.c

```
#include <stdio.h>
#include "xfile.h"

typedef struct Emp
{
   int idEmp;
   char nom[30];
   char dir[50];
   long fecIngreso;
}Emp;

void mostrarArchivo(XFile*);

int main()
{
   int pos;

   // abro el archivo
   XFile f = xopen("EMP.dat","a+b",sizeof(Emp));

   // muestro todos los registros y leo la posicion a eliminar
   mostrarArchivo(&f);
   printf("Ingrese registro a eliminar: ");
   scanf("%d",&pos);

   // mientras no quiera finalizar...
   while( pos>=0 )
   {
      // elimino el registro indicado
      xdelete(&f,pos);

      // muestro todos los registros (que quedan)...
      mostrarArchivo(&f);
      printf("Ingrese registro a eliminar: ");
      scanf("%d",&pos);
   }

   // cierro el archivo
   xclose(&f);

   return 0;
}

void mostrarArchivo(XFile* f)
{
   Emp reg;
   int i=0;

   xreset(f,0);
   xread(f,&reg);
```

```
   while( !xeof(f) )
   {
      printf("[%d] - %d, %s\n",i++,reg.idEmp,reg.nom);
      xread(f,&reg);
   }
}
```

9.4 Resumen

En este capítulo, estudiamos el concepto de “capas de abstracción” y vimos que el TAD es la herramienta fundamental para implementarlo correctamente dentro de los límites que marca el paradigma de la programación estructurada. Más adelante, cuando estudiemos clases y objetos, veremos que las clases reemplazan al TAD.

En el próximo capítulo, analizaremos ejercicios integradores cuya resolución requerirá aplicar todos los temas estudiados hasta aquí.

9.5 Contenido de la página Web de apoyo

El material marcado con asterisco (*) solo está disponible para docentes.

9.5.1 Mapa conceptual

9.5.2 Autoevaluaciones

9.5.3 Presentaciones*