Archivos en C/C++

Almacenar y recuperar información

Salvador Pozo

http://conclase.net

1 Generalidades

Muy a menudo necesitamos almacenar cierta cantidad de datos de forma más o menos permanente. La memoria del ordenador es volatil, y lo que es peor, escasa y cara. De modo que cuando tenemos que guardar nuestros datos durante cierto tiempo tenemos que recurrir a sistemas de almacenamiento más económicos, aunque sea a costa de que sean más lentos.

Durante la historia de los ordenadores se han usado varios métodos distintos para el almacenamiento de datos. Al principio se recurrió a cintas de papel perforadas, después a tarjetas perforadas. Más adelante se pasó al soporte magnético, empezando por grandes rollos de cintas magnéticas abiertas.

Hasta aquí, todos los sistemas de almacenamiento externo eran secuenciales, es decir, no permitían acceder al punto exacto donde se guardaba la información sin antes haber partido desde el principio y sin haber leído toda la información, hasta el punto donde se encontrase la que estábamos buscando.

Con las cintas magnéticas empezó lo que con el tiempo sería el acceso aleatorio a los datos. Se podía reservar parte de la cinta para guardar cierta información sobre la situación de los datos, y añadir ciertas marcas que hicieran más sencillo localizarla.

Pero no fué hasta la aparición de los discos magnéticos cuando ésta técnica llegó a su sentido más amplio. En los discos es más sencillo acceder a cualquier punto de la superficie en poco tiempo, ya que se accede al punto de lectura y escritura usando dos coordenadas físicas. Por una parte la cabeza de lectura/escritura se puede mover en el sentido del radio del disco, y por otra el disco gira permanentemente, con lo que cualquier punto del disco pasa por la cabeza en un tiempo relativamente corto. Esto no pasa con las cintas, donde sólo hay una coordenada física.

Con la invención y proliferación de los discos se desarrollaron los ficheros de acceso aleatorio, que permiten acceder a cualquier dato almacenado en un fichero en relativamente poco tiempo.

Actualmente, los discos duros tienen una enorme capacidad y son muy rápidos, aunque aún siguen siendo lentos, en comparación con las memorias RAM. El caso de los CD es algo intermedio. En realidad son secuenciales en cuanto al modo de guardar los datos, cada disco sólo tiene una pista de datos grabada en espiral. Sin embargo, este sistema, combinado con algo de memoria RAM, proporciona un acceso muy próximo al de los discos duros.

En cuanto al tipo de acceso, en C y C++ podemos clasificar los archivos según varias categorías:

1. Dependiendo de la dirección del flujo de datos:
    - De entrada: los datos se leen por el programa desde el archivo.
    - De salida: los datos se escriben por el programa hacia el archivo.
    - De entrada/salida: los datos pueden se escritos o leídos.
2. Dependiendo del tipo de valores permitidos a cada byte:
    - De texto: sólo están permitidos ciertos rangos de valores para cada byte. Algunos bytes tienen un significado especial, por ejemplo, el valor hexadecimal 0x1A marca el fin de fichero. Si abrimos un archivo en modo texto, no será posible leer más allá de un byte con ese valor, aunque el fichero sea más largo.
    - Binarios: están permitidos todos lo valores para cada byte. En estos archivos el final del fichero se detecta de otro modo, dependiendo del soporte y del sistema operativo. La mayoría de las veces se hace guardando la longitud del fichero. Cuando queramos almacenar valores enteros, o en coma flotante, o imágenes, etc, deberemos usar este tipo de archivos.
3. Según el tipo de acceso:
    - Archivos secuenciales: imitan el modo de acceso de los antiguos ficheros secuenciales almacenados en cintas

magnéticas y
    - Archivos de acceso aleatorio: permiten acceder a cualquier punto de ellos para realizar lecturas y/o escrituras.
4. Según la longitud de registro:
    - Longitud variable: en realidad, en este tipo de archivos no tiene sentido hablar de longitud de registro, podemos considerar cada byte como un registro. También puede suceder que nuestra aplicación conozca el tipo y longitud de cada dato almacenado en el archivo, y lea o escriba los bytes necesarios en cada ocasión. Otro caso es cuando se usa una marca para el final de registro, por ejemplo, en ficheros de texto se usa el carácter de retorno de línea para eso. En estos casos cada registro es de longitud diferente.
    - Longitud constante: en estos archivos los datos se almacenan en forma de registro de tamaño constante. En C usaremos estructuras para definir los registros. C dispone de funciones de biblioteca adecuadas para manejar este tipo de ficheros.
    - Mixtos: en ocasiones pueden crearse archivos que combinen los dos tipos de registros, por ejemplo, dBASE usa registros de longitud constante, pero añade un registro especial de cabecera al principio para definir, entre otras cosas, el tamaño y el tipo de los registros.

Es posible crear archivos combinando cada una de estas categorías, por ejemplo: archivos secuenciales de texto de longitud de registro variable, que son los típicos archivos de texto. Archivos de acceso aleatorio binarios de longitud de registro constante, normalmente usados en bases de datos. Y también cualquier combinación menos corriente, como archivos secuenciales binarios de longitud de registro constante, etc.

En cuanto a cómo se definen estas propiedades, hay dos casos. Si son binarios o de texto o de entrada, salida o entrada/salida, se define al abrir el fichero, mediante la función fopen en C o mediante el método open de fstream en C++.

La función open usa dos parámetros. El primero es el nombre del fichero que contiene el archivo. El segundo es em modo que es una cadena que indica el modo en que se abrirá el archivo: lectura o escritura, y el tipo de datos que contiene: de texto o binarios.

En C, los ficheros admiten seis modos en cuanto a la dirección del flujo de datos:

- r: sólo lectura. El fichero debe existir.
- w: se abre para escritura, se crea un fichero nuevo o se sobrescribe si ya existe.
- a: añadir, se abre para escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.
- r+: lectura y escritura. El fichero debe existir.
- w+: lectura y escritura, se crea un fichero nuevo o se sobrescribe si ya existe.
- a+: añadir, lectura y escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.

En cuanto a los valores permitidos para los bytes, se puede añadir otro carácter a la cadena de modo:

- t: modo texto. Normalmente es el modo por defecto. Se suele omitir.
- b: modo binario.

En ciertos sistemas operativos no existe esta distinción, y todos los ficheros son binarios.

En C++ es algo diferente, el constructor de las clases ifstream, ofstream y fstream admite los parámetros para abrir el fichero directamente, y también disponemos del método open, para poder crear el stream sin asociarlo con un fichero concreto y hacer esa asociación más tarde.

Buffers

Ya hemos comentado que el acceso a los ficheros es lento, y lo es mucho, comparado con el acceso a memoria. Es por eso que, generalmente, no se accede a ficheros externos cada vez que se realiza una operación de lectura o escritura.

En su lugar, se mantiene una copia de una parte del fichero en la memoria, se realizan las operaciones de lectura/escritura que sea posible dentro de esa zona, y cuando sea necesario, porque alguna operación acceda a posiciones fuera de la zona almacenada, se vuelca esa zona al fichero y se lee otro tramo del fichero en memoria.

A estas zonas se le llaman *buffers*, y mejoran sensiblemente el acceso a los ficheros en lo que respecta a la velocidad.

Cuanto más grande es un buffer, mejor será el tiempo de acceso al fichero. En el caso ideal, el tamaño del buffer es mayor o igual que el del fichero, y todas las operaciones de lectura y escritura del fichero se realizan en memoria, de modo que sólo es necesario hacer una lectura del fichero y, si se ha modificado, una escritura.

Pero no todo son ventajas. Cuando se trabaja con buffers, las actualizaciones físicas del fichero están diferidas, en relación a las actualizaciones hechas por el programa, de modo que el fichero no siempre tiene una información actualizada.

Esto plantea dos problemas:

- Si la aplicación termina de forma inesperada, por un error o por una avería, el contenido del buffer modificado no se almacenará en el fichero, y su estructura puede quedar corrupta y los datos inservibles.
- Cuando un fichero deba ser accedido por varios usuarios de forma simultánea, se pueden presentar problemas de concurrencia. Por ejemplo, un usuario lee una parte del fichero en su buffer, y modifica su contenido. Mientras tanto, otro usuario, desde otra máquina, accede al mismo fichero, y a la misma zona, pero el contenido no está actualizado con las modificaciones realizadas por el primer usuario. El peligro es mayor si los dos están haciendo modificaciones en el mismo

fichero, ya que es posible que las modificaciones realizadas por un usuario queden anuladas por las que ha hecho el otro.

El primer caso puede minimizarse, aunque no evitarse siempre, si se guarda el contenido del buffer antes de realizar operaciones potencialmente peligrosas. Aunque nada puede evitar la corrupción de ficheros en caso de avería.

El segundo caso requiere protecciones por parte del sistema operativo o de las aplicaciones que accedan a ficheros compartidos. Estas protecciones van desde las más simples, como la imposibilidad de que un segundo usuario acceda a un fichero abierto, hasta métodos más sutiles, como bloqueo de ficheros, o partes de ficheros. Estos bloqueos asignan una zona del fichero al primer usuario que lo solicite, e impiden a otros usuarios acceder a la misma zona, aunque no a otras.

Cómo funcionan los discos

Para empezar, los discos distribuyen los datos en dos o tres dimensiones. Las cabezas de lectura/escritura se mueven a lo largo del radio del disco, a distancias preestablecidas. Cada una de esas distancias define una *pista*.

Si la cabeza no se mueve, permanece siempre sobre la misma pista. Dependiendo de la densidad del soporte magnético, las pistas podrán estar más o menos próximas entre si, y por lo tanto, en la misma superficie se podrán almacenar más o menos pistas.

A su vez, cada pista está dividida en partes más pequeñas, llamadas *sectores*. Cada sector puede almacenar un número determinado de bytes, y de nuevo, dependiendo de la densidad del soporte, cada pista se podrá dividir en más o menos sectores.

Por último, en el caso de los discos duros, cada disco está compuesto en realidad por varios discos llamados *platos*, cada plato tiene dos caras, y en cada cara se coloca una cabeza de lectura/escritura.

De modo que para acceder a un dato será necesario calcular en qué plato, pista y sector está almacenado, mover la cabeza a la pista adecuada y esperar a que el sector pase por debajo de la cabeza correspondiente al plato indicado.

El tiempo de acceso depende de la capacidad de la cabeza para localizar la pista, del número de sectores por pista (cuantos más haya, mayor será el promedio de tiempo necesario para que un sector pase bajo la cabeza), y de la velocidad de giro del disco.

Parece sencillo localizar una pista, pero no tanto localizar un sector. A fin de cuentas, el disco puede empezar a girar en cualquier posición, y no es posible distinguir donde estaban los sectores la última vez que el disco estuvo girando. Además, en los discos duros actuales se aprovecha mejor el espacio haciendo que las pistas exteriores, con mayor circunferencia, se dividan en más sectores que las interiores. Esto permite aprovechar mejor la densidad del disco, que por supuesto, es uniforme.

Para que sea posible localizar la información en un disco hay que almacenar ciertas marcas en él. Al conjunto de esas marcas se le llama *formato* y a la acción de hacer esas marcas, se le llama *formatear* el disco.

En los discos magnéticos, las marcas se almacenan del mismo modo que los datos: mediante campos magnéticos creados por la cabeza de lectura/escritura. En cada pista se almacenan ciertas marcas que indican donde empieza cada sector y que identifican cada una de las pistas y sectores. Esto, por cierto, disminuye el espacio disponible para los datos. Antiguamente se distinguía entre la capacidad *bruta* del disco, sin descontar el espacio destinado al formato, y la capacidad *útil*, que es la que nos interesa en realidad.

Otros soportes, como algunos disquetes primitivos, tenían orificios en la superficie del disco que podían ser detectados ópticamente, aunque esto no evitaba que el disco tuviese que tener un formato.

En los discos magneto-ópticos las escrituras se hacen de forma magnética, sobre un disco con dos capas calentado mediante un

láser de alta densidad, mientras que las lecturas se hacen de forma óptica, mediante un láser de baja densidad. De modo que estos discos permiten mayor capacidad de almacenamiento, y permiten hacer lecturas muchos más rápidas, aunque la escritura es más lenta.

En los discos duros la estructura es más complicada, ya que existe más de un disco, y las cabezas se sitúan en ambas caras de cada disco. Sin embargo, todas las cabezas se mueven de forma simultánea, aunque las lecturas y escrituras sólo se hacen en una superficie a la vez. Esto hace que el acceso sea más rápido cuantos más platos existan.

La unidad mínima que se puede leer o escribir en un disco es un sector. El tamaño del sector es variable, generalmente son de 512 bytes, pero pueden ser diferentes. El sistema operativo no trabaja directamente con sectores, sino con *clusters*. Cada cluster tiene un número entero de sectores.

Los clusters son una unidad lógica, no física. En principio se crearon cuando la capacidad de los discos creció hasta el punto que con los protoclos de 16 bits no era posible direccionar todos los sectores. Agrupando sectores seguía siendo posible aprovechar toda la capacidad del disco. Esto volvió a suceder con los protocolos de 32 bits.

Generalmente es mejor que el tamaño de los clusters sea pequeño, ya que de ese modo se aprovecha mejor el espacio de almacenamiento del disco. Si por ejemplo, en un disco de 100KB almacenamos sólo ficheros de 1KB, en teoría podríamos almacenar 100 ficheros. Pero si el tamaño del cluster es de 2KB, el número máximo de ficheros será 50, y si el tamaño del cluster es de 16Kb, sólo podremos almacenar 6. De hecho, en un disco con clusters de 16KB, un fichero de un byte ocupará el mismo espacio que uno de 16KB.

Ficheros que cambian de tamaño

Por supuesto, todo lo explicado anteriormente es muy simple cuando sólo hacemos lecturas o cuando las escrituras no implican que el fichero deba ser más largos o más cortos.

Físicamente, los ficheros sólo pueden crecer por el extremo final. Esto es evidente en el caso de ficheros secuenciales, donde cualquier escritura que no se realize al final implica la sobrescritura de datos previos.

Las limitaciones físicas de los soportes de datos explican por qué no se pueden insertar datos en el interior de un fichero. En teoría se podría insertar un bloque de datos cuyo tamaño sea múltiplo del tamaño del cluster. De ese modo se podrían insertar nuevos clusters dentro del fichero. Pero esto es un caso muy especial, y nunca se hace.

Lo que se hace en realidad es mover el resto del fichero hacia adelante para dejar espacio para la nueva información, o en caso de borrar datos, mover hacia atrás, de modo que se sobrescriban los datos eliminados. De modo que si hay que añadir un byte en la primera posición de un fichero, esto implica que se ha de copiar todo el fichero.

2 Tipos, funciones y clases usados frecuentemente con ficheros

Funciones y tipos C estándar:

Tipo FILE

C define la estructura de datos FILE en el fichero de cabecera stdio.h para el manejo de ficheros. Nosotros siempre usaremos punteros a estas estructuras.

La definición de ésta estructura depende del compilador, pero en general mantienen un campo con la posición actual de lectura/escritura, un buffer para mejorar las prestaciones de acceso al fichero y algunos campos para uso interno.

Función fopen

Sintaxis:

```
FILE *fopen(char *nombre, char *modo);
```

Esta función sirve para abrir y crear ficheros en disco. El valor de retorno es un puntero a una estructura FILE. Los parámetros de entrada son:

1. nombre: una cadena que contiene un nombre de fichero válido, esto depende del sistema operativo que estemos usando. El nombre puede incluir el camino completo.

2. modo: especifica en tipo de fichero que se abrirá o se creará y el tipo de datos que puede contener, de texto o binarios:
    - r: sólo lectura. El fichero debe existir.
    - w: se abre para escritura, se crea un fichero nuevo o se sobreescribe si ya existe.
    - a: añadir, se abre para escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.
    - r+: lectura y escritura. El fichero debe existir.
    - w+: lectura y escritura, se crea un fichero nuevo o se sobreescribe si ya existe.
    - a+: añadir, lectura y escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.
    - t: tipo texto, si no se especifica "t" ni "b", se asume por defecto que es "t"
    - b: tipo binario.

Función fclose

Sintaxis:

```
int fclose(FILE *fichero);
```

Es importante cerrar los ficheros abiertos antes de abandonar la aplicación. Esta función sirve para eso. Cerrar un fichero almacena los datos que aún están en el buffer de memoria, y actualiza algunos datos de la cabecera del fichero que mantiene el sistema operativo. Además permite que otros programas puedan abrir el fichero para su uso. Muy a menudo, los ficheros no pueden ser compartidos por varios programas.

Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF. El parámetro es un puntero a la estructura FILE del fichero que queremos cerrar.

Función fgetc

Sintaxis:

```
int fgetc(FILE *fichero);
```

Esta función lee un carácter desde un fichero.

El valor de retorno es el carácter leído como un **unsigned char** convertido a **int**. Si no hay ningún carácter disponible, el valor de retorno es EOF. El parámetro es un puntero a una estructura FILE del fichero del que se hará la lectura.

Función fputc

Sintaxis:

```
int fputc(int caracter, FILE *fichero);
```

Esta función escribe un carácter a un fichero.

El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será EOF. Los parámetros de entrada son el carácter a escribir, convertido a **int** y un puntero a una estructura FILE del fichero en el que se hará la escritura.

Función feof

Sintaxis:

```
int feof(FILE *fichero);
```

Esta función sirve para comprobar si se ha alcanzado el final del fichero. Muy frecuentemente deberemos trabajar con todos los valores almacenados en un archivo de forma secuencial, la forma que suelen tener los bucles para leer todos los datos de un archivo es permanecer leyendo mientras no se detecte el fin de fichero. Esta

función suele usarse como prueba para verificar si se ha alcanzado o no ese punto.

El valor de retorno es distinto de cero sólo si no se ha alcanzado el fin de fichero. El parámetro es un puntero a la estructura FILE del fichero que queremos verificar.

Función rewind

Sintaxis:

```
void rewind(FILE *fichero)
```

Es una función heredada de los tiempos de las cintas magnéticas. Literalmente significa "rebobinar", y hace referencia a que para volver al principio de un archivo almacenado en cinta, había que rebobinarla. Eso es lo que hace ésta función, sitúa el cursor de lectura/escritura al principio del archivo.

El parámetro es un puntero a la estructura FILE del fichero que queremos rebobinar.

Ejemplos:

```
// ejemplo1.c: Muestra un fichero dos veces.
#include <stdio.h>

int main()
{
   FILE *fichero;

   fichero = fopen("ejemplo1.c", "r");
   while(!feof(fichero)) fputc(fgetc(fichero), stdout);
   rewind(fichero);
   while(!feof(fichero)) fputc(fgetc(fichero), stdout);
   fclose(fichero);
   getchar();
   return 0;
}
```

Función fgets

Sintaxis:

```
char *fgets(char *cadena, int n, FILE *fichero);
```

Esta función está diseñada para leer cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.

El parámetro *n* nos permite limitar la lectura para evitar derbordar el espacio disponible en la *cadena*.

El valor de retorno es un puntero a la cadena leída, si se leyó con éxito, y es NULL si se detecta el final del fichero o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un puntero a una estructura FILE del fichero del que se leerá.

Función fputs

Sintaxis:

```
int fputs(const char *cadena, FILE *stream);
```

La función fputs escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final.

El valor de retorno es un número no negativo o EOF en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura FILE del fichero donde se realizará la escritura.

Función fread

Sintaxis:

```
size_t fread(void *puntero, size_t tamaño, size_t
nregistros, FILE *fichero);
```

Esta función está pensada para trabajar con registros de longitud constante. Es capaz de leer desde un fichero uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. El usuario es responsable de asegurarse de que hay espacio suficiente para contener la información leída.

El valor de retorno es el **número de registros leídos**, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.

Función fwrite

Sintaxis:

```
size_t fwrite(void *puntero, size_t tamaño, size_t
nregistros, FILE *fichero);
```

Esta función también está pensada para trabajar con registros de longitud constante y forma pareja con fread. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada.

El valor de retorno es el **número de registros escritos**, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.

Ejemplo:

```
// copia.c: Copia de ficheros
```

```
// Uso: copia <fichero_origen> <fichero_destino>

#include <stdio.h>

int main(int argc, char **argv) {
    FILE *fe, *fs;
    unsigned char buffer[2048]; // Buffer de 2 Kbytes
    int bytesLeidos;

    if(argc != 3) {
       printf("Usar: copia <fichero_origen>
<fichero_destino>\n");
       return 1;
    }

    // Abrir el fichero de entrada en lectura y binario
    fe = fopen(argv[1], "rb");
    if(!fe) {
       printf("El fichero %s no existe o no puede ser
abierto.\n", argv[1]);
       return 1;
    }
    // Crear o sobreescribir el fichero de salida en binario
    fs = fopen(argv[2], "wb");
    if(!fs) {
       printf("El fichero %s no puede ser creado.\n",
argv[2]);
       fclose(fe);
       return 1;
    }
    // Bucle de copia:
    while((bytesLeidos = fread(buffer, 1, 2048, fe)))
       fwrite(buffer, 1, bytesLeidos, fs);
    // Cerrar ficheros:
    fclose(fe);
    fclose(fs);
    return 0;
}
```

Función fprintf

Sintaxis:

```
int fprintf(FILE *fichero, const char *formato, ...);
```

La función fprintf funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un fichero en lugar de a la pantalla.

Función fscanf

Sintaxis:

```
int fscanf(FILE *fichero, const char *formato, ...);
```

La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

Función fflush

Sintaxis:

```
int fflush(FILE *fichero);
```

Esta función fuerza la salida de los datos acumulados en el buffer de salida del *fichero*. Para mejorar las prestaciones del manejo de ficheros se utilizan buffers, almacenes temporales de datos en memoria, las operaciones de salida se hacen a través del buffer, y sólo cuando el buffer se llena se realiza la escritura en el disco y se vacía el buffer. En ocasiones nos hace falta vaciar ese buffer de un modo manual, para eso sirve ésta función.

El valor de retorno es cero si la función se ejecutó con éxito, y EOF si hubo algún error. El parámetro de entrada es un puntero a la estructura FILE del fichero del que se quiere vaciar el buffer. Si es NULL se hará el vaciado de todos los ficheros abiertos.

Funciones C específicas para ficheros de acceso aleatorio

Función fseek

Sintaxis:

```
int fseek(FILE *fichero, long int desplazamiento, int
origen);
```

Esta función sirve para situar el cursor del fichero para leer o escribir en el lugar deseado.

El valor de retorno es cero si la función tuvo éxito, y un valor distinto de cero si hubo algún error.

Los parámetros de entrada son: un puntero a una estructura FILE del fichero en el que queremos cambiar el cursor de lectura/escritura, el valor del desplazamiento y el punto de origen desde el que se calculará el desplazamiento.

El parámetro *origen* puede tener tres posibles valores:

1. SEEK_SET el desplazamiento se cuenta desde el principio del fichero. El primer byte del fichero tiene un desplazamiento cero.
2. SEEK_CUR el desplazamiento se cuenta desde la posición actual del cursor.
3. SEEK_END el desplazamiento se cuenta desde el final del fichero.

Función ftell

Sintaxis:

```
long int ftell(FILE *fichero);
```

La función ftell sirve para averiguar la posición actual del cursor de lectura/excritura de un fichero.

El valor de retorno será esa posición, o -1 si hay algún error.

El parámetro de entrada es un puntero a una estructura FILE del fichero del que queremos leer la posición del cursor de lectura/escritura.

Clases para manejar ficheros en C++

Existen tres clases para manejar ficheros: ifstream, ofstream y fstream. La primera está orientada a ficheros de entrada, la segunda a ficheros de salida, y la tercera puede manejar cualquiera de los dos tipos o ficheros de entrada y salida.

Clase ifstream

El constructor está sobrecargado para poder crear streams de varias maneras:

```
ifstream();
ifstream(const char *name, int mode = ios::in,
   int = filebuf::openprot);
```

El primero sólo crea un stream de entrada pero no lo asocia a ningún fichero. El segundo lo crea, lo asocia al fichero con el nombre "name" y lo abre.

Los parámetros son: el nombre del fichero, el modo, que para ifstream es ios::in por defecto. El tercer parámetro se refiere al buffer, y no nos preocupa de momento.

Clase ofstream

Lo mismo pasa con ofstream, salvo que los valores por defecto de los parámetros son diferentes:

```
ofstream();
ofstream(const char *name, int mode = ios::out,
   int = filebuf::openprot);
```

Clase fstream

```
fstream();
fstream(const char *name, int mode = ios::in,
   int = filebuf::openprot);
```

Método open

Todas estas clases disponen además del método "open", para abrir el fichero a lo largo de la ejecución del programa.

```
void open(const char *name, int mode,
   int prot=filebuf::openprot);
```

"name" es el nombre del fichero, mode es el modo en que se abrirá, puede ser uno o una combinación del tipo enumerado open_mode, de la clase "ios":

```
enum open_mode { in, out, ate, app, trunc, nocreate,
   noreplace, binary };
```

Cada uno de los valores se pueden combinar usando el operador de bits OR (|), y significan lo siguiente:

- in: modo de entrada.
- out: modo de salida.
- ate: abre el fichero y sitúa el cursor al final.
- app: modo append, parecido al anterior, pero las operaciones de escritura siempre se hacen al final del fichero.

- trunc: si se aplica a ficheros de salida, se creará el fichero si no existe previamente, o se truncará con un tamaño de 0 bytes, si existe.
- nocreate: impide crear un fichero si no existe, en ese caso, la función falla.
- noreplace: lo ignoro.
- binary: abre el fichero en modo binario.

Los tres últimos modos probablemente no son estándar, y es posible que no existan en muchos compiladores.

Método close

```
void close();
```

Sencillamente, cierra el fichero asociado a un stream.

Operador >>:

Igual que sucede con el stream estándar cout, el operador de flujo de salida >> se puede usar con streams de salida cuando trabajemos con texto.

Operador <<:

Del mismo modo, al igual que sucede con el stream estándar cin, el operador de flujo de entrada << se puede usar con streams de entrada cuando trabajemos con texto.

Método de salida put:

```
ostream& put(char ch);
```

Sirve para cualquier stream de salida, e inserta un carácter en el stream.

Método de entrada get

```
int get();
istream& get(char*, int len, char = '\n');
istream& get(char&);
istream& get(streambuf&, char = '\n');
```

La primera forma no se recomienda y se considera obsoleta, lee un carácter desde el stream de entrada.

La segunda lee caracteres y los almacena en el buffer indicado en el primer parámetro hasta que se leen "len" caracteres o hasta que se encuentra el carácter indicado en el tercer parámetro, que por defecto es el retorno de línea.

La tercera forma extrae un único carácter en la referencia a char proporcionada.

La cuarta no nos interesa de momento.

Método de entrada getline

```
istream& getline(char*, int, char = '\n');
```

Extrae caracteres hasta que se encuentra el delimitador y los coloca en el buffer, elimina el delimitador del stream de entrada y no lo añade al buffer.

Método eof

```
int eof();
```

Verifica si se ha alcanzado el final del fichero, devuelve un valor nulo si no es así.

Método clear

```
void clear(iostate state=0);
```

Cada vez que se produzca una condición de error en un stream es necesario eliminarla, ya que en caso contrario ninguna operación que se realice sobre él tendrá éxisto. Por ejemplo, si llegamos hasta el final de fichero, el stream quedará en estado "eof" hasta que se elimine explícitamente ese estado. Eso se hace mediante el método "clear", sin parámetros dejará el estado en 0, es decir, sin errores.

Los estados posibles se definen en un enumerado:

```
enum io_state { goodbit, eofbit, failbit, badbit };
```

- goodbit: indica que el estado es correcto.
- eofbit: indica que se ha detectado fin de fichero.
- failbit: indica que una operación sobre el stream ha fallado.
- badbit: se activa si falla una operación de escritura de buffers.

Método bad

```
int bad();
```

Devuelve el estado del bit "badbit".

Método fail:

```
int fail();
```

Devuelve el estado del bit "failbit".

Método good

```
int good();
```

Devuelve el estado del bit "goodbit".

Ejemplo:

Veamos el ejemplo anterior de mostrar dos veces un fichero, pero esta vez escrito para C++ usando streams:

```
// ejemplo1.cpp: Muestra un fichero dos veces.
#include <iostream>
#include <fstream>
using namespace std;

int main() {
   ifstream fichero("ejemplo1.cpp");
   char c;

   while(fichero.get(c)) cout.put(c);
   fichero.clear(); // (1)
   fichero.seekg(0);
   while(fichero.get(c)) cout.put(c);
   fichero.close();
   cin.get();
   return 0;
}
```

Como vemos en (1), es necesario eliminar el bit de eof, que se ha activado al leer hasta el final del fichero, cuando el último intento de llamar a "get" ha fallado, porque se ha terminado el fichero.

Método is_open

```
int is_open();
```

Devuelve un valor no nulo si el fichero está abierto.

Método flush

```
ostream& flush();
```

Realiza las operaciones de escritura pendientes que aún se han realizado sólo en el buffer.

Métodos relacionados con acceso aleatorio

Disponemos de otro tipo enumerado en ios para indicar movimientos relativos dentro de un stream de acceso aleatorio:

```
enum seek_dir { beg, cur, end};
```

- beg: relativo al principio del fichero.
- cur: relativo a la posición actual del cursor dentro del fichero.
- end: relativo al final del fichero.

Método seekg

Cambia la posición del cursor en streams de entrada.

```
istream& seekg(streampos pos);
istream& seekg(streamoff offset, seek_dir dir);
```

La primera forma es para cambiar la posición de modo absoluto. La segunda para cambios relativos, en la que se indica el salto en el primer parámetro y el punto de partida en el segundo, que puede ser cualquiera de los indicados anteriormente: ios::beg, ios::cur o ios::end.

Método seekp

Cambia la posición del cursor en streams de salida.

```
ostream& seekp(streampos pos);
ostream& seekp(streamoff offset, seek_dir);
```

Lo mismo que seekg, pero aplicado a estream de salida.

Método tellg

```
streampos tellg();
```

Devuelve la posición actual del cursor dentro de un stream de entrada.

Método tellp

```
streampos tellp();
```

Devuelve la posición actual del cursor dentro de un stream de salida.

Método read

```
istream& read(char*, int);
```

Lee el número de caracteres indicado en el segundo parámetro dendro del buffer suministrado por el primero.

Método gcount

```
int gcount();
```

Devuelve el número de caracteres sin formato de la última lectura. Las lecturas sin formato son las realizadas mediante las funciones get, getline y read.

Método write

```
ostream& write(const char*, int);
```

Escribe el número de caracteres indicado en el segundo parámetro desde el buffer suministrado por el primero.

Ejemplo:

De nuevo haremos el ejemplo de copiar ficheros, pero esta vez usando streams.

```
// copia.cpp: Copia de ficheros
// Uso: copia <fichero_origen> <fichero_destino>

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv) {
    ifstream entrada;
    ofstream salida;

    char buffer[2048]; // Buffer de 2 Kbytes
    int bytesLeidos;

    if(argc != 3) {
       printf("Usar: copia <fichero_origen>
<fichero_destino>\n");
       return 1;
    }

    // Abrir el fichero de entrada en lectura y binario
    entrada.open(argv[1]);
```

```
    if(!entrada.good()) {
       printf("El fichero %s no existe o no puede ser
abierto.\n", argv[1]);
       return 1;
    }
    // Crear o sobreescribir el fichero de salida en binario
    salida.open(argv[2]);
    if(!salida.good()) {
       printf("El fichero %s no puede ser creado.\n",
argv[2]);
       entrada.close();
       return 1;
    }
    // Bucle de copia:
    do {
       entrada.read(buffer, 2048);
       bytesLeidos = entrada.gcount();
       salida.write(buffer, bytesLeidos);
    } while(bytesLeidos > 0);
    // Cerrar ficheros:
    entrada.close();
    salida.close();
    return 0;
}
```

3 Archivos secuenciales

En estos archivos, la información sólo puede leerse y escribirse empezando desde el principio del archivo.

Los archivos secuenciales tienen algunas características que hay que tener en cuenta:

1. La escritura de nuevos datos siempre se hace al final del archivo.
2. Para leer una zona concreta del archivo hay que avanzar siempre, si la zona está antes de la zona actual de lectura, será necesario "rebobinar" el archivo.
3. Los ficheros sólo se pueden abrir para lectura o para escritura, nunca de los dos modos a la vez.

Esto es en teoría, por supuesto, en realidad C no distingue si los archivos que usamos son secuenciales o no, es el tratamiento que hagamos de ellos lo que los clasifica como de uno u otro tipo.

Pero hay archivos que se comportan siempre como secuenciales, por ejemplo los ficheros de entrada y salida estándar: stdin, stdout, stderr y stdaux.

Tomemos el caso de stdin, que suele ser el teclado. Nuestro programa sólo podrá abrir ese fichero como de lectura, y sólo podrá leer los caracteres a medida que estén disponibles, y en el mismo orden en que fueron tecleados.

Lo mismo se aplica para stdout y stderr, que es la pantalla, en estos casos sólo se pueden usar para escritura, y el orden en que se muestra la información es el mismo en que se envía.

Un caso especial es stdaux, que suele ser el puerto serie. También es un archivo secuencial, con respecto al modo en que se leen y escriben los datos. Sin embargo se un fichero de entrada y salida.

Trabajar con archivos secuenciales tiene algunos inconvenientes. Por ejemplo, imagina que tienes un archivo de este tipo en una cinta magnética. Por las características físicas de este soporte, es eviente que sólo podemos tener un fichero abierto en cada unidad de cinta. Cada fichero puede ser leído, y también sobrescrito, pero en general, los archivos que haya a continuación del que escribimos se perderán, o bien serán sobreescritos al crecer el archivo, o quedará un espacio vacío entre el final del archivo y el principio del siguiente.

Lo normal cuando se quería actualizar el contenido de un archivo de cinta añadiendo o modificando datos, era abrir el archivo en modo lectura en una unidad de cinta, y crear un nuevo fichero de escritura en una unidad de cinta distinta. Los datos leídos de una cinta se editan o modifican, y se copian en la otra secuencialmente.

Cuando trabajemos con archivos secuenciales en disco haremos lo mismo, pero en ese caso no necesitamos dos unidades de disco, ya que en los discos es posible abrir varios archivos simultaneamente.

En cuanto a las ventajas, los archivos secuenciales son más sencillos de manejar, ya que requieren menos funciones, además son más rápidos, ya que no permiten moverse a lo largo del archivo, el punto de lectura y escritura está siempre determinado.

En ocasiones pueden ser útiles, por ejemplo, cuando sólo se quiere almacenar cierta información a medida que se recibe, y no interesa analizarla en el momento. Posteriormente, otro programa puede leer esa información desde el principio y analizarla. Este es el caso de archivos "log" o "diarios" por ejemplo, los servidores de las páginas WEB pueden generar una línea de texto cada vez que alguien accede al una de las páginas y las guardan en un fichero secuencial.

4 Archivos de acceso aleatorio

Los archivos de acceso aleatorio son más versátiles, permiten acceder a cualquier parte del fichero en cualquier momento, como si fueran arrays en memoria. Las operaciones de lectura y/o escritura pueden hacerse en cualquier punto del archivo.

En general se suelen establecer ciertas normas para la creación, aunque no todas son obligatorias:

1. Abrir el archivo en un modo que te permita leer y escribir. Esto no es imprescindible, es posible usar archivos de acceso aleatorio sólo de lectura o de escritura.
2. Abrirlo en modo binario, ya que algunos o todos los campos de la estructura pueden no ser caracteres.
3. Usar funciones como fread y fwrite, que permiten leer y escribir registros de longitud constante desde y hacia un fichero.
4. Usar la función fseek para situar el puntero de lectura/escritura en el lugar apropiado de tu archivo.

Por ejemplo, supongamos que nuestros registros tienen la siguiente estructura:

```
struct stRegistro {
   char Nombre[34];
   int dato;
   int matriz[23];
} reg;
```

Teniendo en cuenta que los registros empiezan a contarse desde el cero, para hacer una lectura del registro número 6 usaremos:

```
fseek(fichero, 5*sizeof(stRegistro), SEEK_SET);
```

```
fread(&reg, sizeof(stRegistro), 1, fichero);
```

Análogamente, para hacer una operación de escritura, usaremos:

```
fseek(fichero, 5*sizeof(stRegistro), SEEK_SET);
fwrite(&reg, sizeof(stRegistro), 1, fichero);
```

Muy importante: después de cada operación de lectura o escritura, el cursor del fichero se actualiza automáticamente a la siguiente posición, así que es buena idea hacer siempre un fseek antes de un fread o un fwrite.

En el caso de streams, la forma de trabajar es análoga:

```
fichero.seekg(5*sizeof(stRegistro), ios::beg);
fichero.read(&reg, sizeof(stRegistro));
```

Y para hacer una operación de escritura, usaremos:

```
fichero.seekp(5*sizeof(stRegistro), ios::beg);
fichero.write(&reg, sizeof(stRegistro));
```

Calcular la longitud de un fichero

Para calcular el tamaño de un fichero, ya sea en bytes o en registros se suele usar el siguiente procedimiento:

```
long nRegistros;
long nBytes;
fseek(fichero, 0, SEEK_END); // Colocar el cursor al final
del fichero
nBytes = ftell(fichero); // Tamaño en bytes
```

```
nRegistros = ftell(fich)/sizeof(stRegistro); // Tamaño en
registros
```

En el caso de streams:

```
long nRegistros;
long nBytes;
fichero.seekg(0, ios::end); // Colocar el cursor al final
del fichero
nBytes = fichero.tellg(); // Tamaño en bytes
nRegistros = fichero.tellg()/sizeof(stRegistro); // Tamaño
en registros
```

Borrar registros

Borrar registros puede ser complicado, ya que no hay ninguna función de biblioteca estándar que lo haga.

Es su lugar se suele usar uno de estos dos métodos:

1. Marcar el registro como borrado o no válido, para ello hay que añadir un campo extra en la estructura del registro:

```
struct stRegistro {
   char Valido;  // Campo que indica si el registro es
válido
   char Nombre[34];
   int dato;
   int matriz[23];
};
```

Si el campo *Valido* tiene un valor prefijado, por ejemplo 'S' o ' ', el registro es válido. Si tiene un valor prefijado, por ejemplo 'N' o '*', el registro será inválido o se considerará borrado.

De este modo, para borrar un registro sólo tienes que cambiar el valor de ese campo.

Pero hay que tener en cuenta que será el programa el encargado de tratar los registros del modo adecuado dependiendo del valor del campo *Valido*, el hecho de **marcar un registro no lo borra físicamente**.

Si se quiere elaborar más, se puede mantener un fichero auxiliar con la lista de los registros borrados. Esto tiene un doble propósito:

- Que se pueda diseñar una función para sustituir a fseek() de modo que se tengan en cuenta los registros marcados.
- Que al insertar nuevos registros, se puedan sobrescribir los anteriormente marcados como borrados, si existe alguno.

2. Hacer una copia del fichero en otro fichero, pero sin copiar el registro que se quiere borrar. Este sistema es más tedioso y lento, y requiere cerrar el fichero y borrarlo o renombrarlo, antes de poder usar de nuevo la versión con el registro eliminado.

Lo normal es hacer una combinación de ambos, durante la ejecución normal del programa se borran registros con el método de marcarlos, y cuando se cierra la aplicación, o se detecta que el porcentaje de registros borrados es alto o el usuario así lo decide, se "empaqueta" el fichero usando el segundo método.

Ejemplo

A continuación se incluye un ejemplo de un programa que trabaja con registros de acceso aleatorio, es un poco largo, pero bastante completo:

```
// alea.c: Ejemplo de ficheros de acceso aleatorio.
#include <stdio.h>
#include <stdlib.h>

struct stRegistro {
   char valido;  // Campo que indica si el registro es
válido S->Válido, N->Inválido
   char nombre[34];
```

```
   int dato[4];
};

int Menu();
void Leer(struct stRegistro *reg);
void Mostrar(struct stRegistro *reg);
void Listar(long n, struct stRegistro *reg);
long LeeNumero();
void Empaquetar(FILE **fa);

int main()
{
   struct stRegistro reg;
   FILE *fa;
   int opcion;
   long numero;
   fa = fopen("alea.dat", "r+b");          // Este modo
permite leer y escribir
   if(!fa) fa = fopen("alea.dat", "w+b");  // si el fichero
no existe, lo crea.
   do {
      opcion = Menu();
      switch(opcion) {
         case '1': // Añadir registro
            Leer(&reg);
            // Insertar al final:
            fseek(fa, 0, SEEK_END);
            fwrite(&reg, sizeof(struct stRegistro), 1, fa);
            break;
         case '2': // Mostrar registro
            system("cls");
            printf("Mostrar registro: ");
            numero = LeeNumero();
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
            fread(&reg, sizeof(struct stRegistro), 1, fa);
            Mostrar(&reg);
            break;
         case '3': // Eliminar registro
            system("cls");
            printf("Eliminar registro: ");
            numero = LeeNumero();
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
            fread(&reg, sizeof(struct stRegistro), 1, fa);
            reg.valido = 'N';
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
```

```
            fwrite(&reg, sizeof(struct stRegistro), 1, fa);
            break;
         case '4': // Mostrar todo
            rewind(fa);
            numero = 0;
            system("cls");
            printf("Nombre
Datos\n");
            while(fread(&reg, sizeof(struct stRegistro), 1,
fa))
               Listar(numero++, &reg);
            system("PAUSE");
            break;
         case '5': // Eliminar marcados
            Empaquetar(&fa);
            break;
      }
   } while(opcion != '0');
   fclose(fa);
   return 0;
}

// Muestra un menú con las opciones disponibles y captura
una opción del usuario
int Menu()
{
   char resp[20];
   do {
      system("cls");
      printf("MENU PRINCIPAL\n");
      printf("--------------\n\n");
      printf("1- Insertar registro\n");
      printf("2- Mostrar registro\n");
      printf("3- Eliminar registro\n");
      printf("4- Mostrar todo\n");
      printf("5- Eliminar registros marcados\n");
      printf("0- Salir\n");
      fgets(resp, 20, stdin);
   } while(resp[0] < '0' && resp[0] > '5');
   return resp[0];
}

// Permite que el usuario introduzca un registro por
pantalla
void Leer(struct stRegistro *reg)
{
   int i;
   char numero[6];
```

```
   system("cls");
   printf("Leer registro:\n\n");
   reg->valido = 'S';
   printf("Nombre: ");
   fgets(reg->nombre, 34, stdin);
   // la función fgets captura el retorno de línea, hay que
eliminarlo:
   for(i = strlen(reg->nombre)-1; i && reg->nombre[i] < ' ';
i--)
      reg->nombre[i] = 0;
   for(i = 0; i < 4; i++) {
      printf("Dato[%1d]: ", i);
      fgets(numero, 6, stdin);
      reg->dato[i] = atoi(numero);
   }
}

// Muestra un registro en pantalla, si no está marcado como
borrado
void Mostrar(struct stRegistro *reg)
{
   int i;
   system("cls");
   if(reg->valido == 'S') {
      printf("Nombre: %s\n", reg->nombre);
      for(i = 0; i < 4; i++) printf("Dato[%1d]: %d\n", i,
reg->dato[i]);
   }
   system("PAUSE");
}

// Muestra un registro por pantalla en forma de listado,
// si no está marcado como borrado
void Listar(long n, struct stRegistro *reg)
{
   int i;
   if(reg->valido == 'S') {
      printf("[%6ld] %-34s", n, reg->nombre);
      for(i = 0; i < 4; i++) printf(", %4d", reg->dato[i]);
      printf("\n");
   }
}

// Lee un número suministrado por el usuario
long LeeNumero()
{
   char numero[6];
   fgets(numero, 6, stdin);
```

```
   return atoi(numero);
}

// Elimina los registros marcados como borrados
void Empaquetar(FILE **fa)
{
   FILE *ftemp;
   struct stRegistro reg;

   ftemp = fopen("alea.tmp", "wb");
   rewind(*fa);
   while(fread(&reg, sizeof(struct stRegistro), 1, *fa))
      if(reg.valido == 'S')
         fwrite(&reg, sizeof(struct stRegistro), 1, ftemp);
   fclose(ftemp);
   fclose(*fa);
   remove("alea.bak");
   rename("alea.dat", "alea.bak");
   rename("alea.tmp", "alea.dat");
   *fa = fopen("alea.dat", "r+b");
}
```

Y esto es un ejemplo equivalente en C++:

```
// alea.cpp: Ejemplo de ficheros de acceso aleatorio.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>
using namespace std;

// Funciones auxiliares:
int Menu();
long LeeNumero();

// Clase registro.
class Registro {
  public:
   Registro(char *n=NULL, int d1=0, int d2=0, int d3=0, int
d4=0) : valido('S') {
     if(n) strcpy(nombre, n); else strcpy(nombre, "");
     dato[0] = d1;
     dato[1] = d2;
     dato[2] = d3;
```

```
      dato[3] = d4;
   }
   void Leer();
   void Mostrar();
   void Listar(long n);

   const bool Valido() { return valido == 'S'; }
   const char *Nombre() { return nombre; }
  private:
   char valido;  // Campo que indica si el registro es
válido
                 // S->Válido, N->Inválido
   char nombre[34];
   int dato[4];
};

// Implementaciones de clase Registro:
// Permite que el usuario introduzca un registro por
pantalla
void Registro::Leer() {
   system("cls");
   cout << "Leer registro:" << endl << endl;
   valido = 'S';
   cout << "Nombre: ";
   cin.getline(nombre, 34);
   for(int i = 0; i < 4; i++) {
      cout << "Dato[" << i << "]: ";
      dato[i] = LeeNumero();
   }
}

// Muestra un registro en pantalla, si no está marcado como
borrado
void Registro::Mostrar()
{
   system("cls");
   if(Valido()) {
      cout << "Nombre: " << nombre << endl;
      for(int i = 0; i < 4; i++)
         cout << "Dato[" << i << "]: " << dato[i] << endl;
   }
   cout << "Pulsa una tecla";
   cin.get();
}

// Muestra un registro por pantalla en forma de listado,
// si no está marcado como borrado
void Registro::Listar(long n) {
```

```
   int i;

   if(Valido()) {
      cout << "[" << setw(6) << n << "] ";
      cout << setw(34) << nombre;
      for(i = 0; i < 4; i++)
         cout << ", " << setw(4) << dato[i];
      cout << endl;
   }
}

// Clase Datos, almacena y trata los datos.
class Datos :public fstream {
  public:
   Datos() : fstream("alea.dat", ios::in | ios::out |
ios::binary) {
      if(!good()) {
         open("alea.dat", ios::in | ios::out | ios::trunc |
ios::binary);
         cout << "fichero creado" << endl;
         cin.get();
      }
   }
  ~Datos() {
     Empaquetar();
   }
   void Guardar(Registro &reg);
   bool Recupera(long n, Registro &reg);
   void Borrar(long n);

  private:
   void Empaquetar();
};

// Implementación de la clase Datos.
void Datos::Guardar(Registro &reg) {
   // Insertar al final:
   clear();
   seekg(0, ios::end);
   write(reinterpret_cast<char *> (&reg), sizeof(Registro));
   cout << reg.Nombre() << endl;
}

bool Datos::Recupera(long n, Registro &reg) {
   clear();
   seekg(n*sizeof(Registro), ios::beg);
   read(reinterpret_cast<char *> (&reg), sizeof(Registro));
   return gcount() > 0;
```

```
}

// Marca el registro como borrado:
void Datos::Borrar(long n) {
   char marca;

   clear();
   marca = 'N';
   seekg(n*sizeof(Registro), ios::beg);
   write(&marca, 1);
}

// Elimina los registros marcados como borrados
void Datos::Empaquetar() {
   ofstream ftemp("alea.tmp", ios::out);
   Registro reg;

   clear();
   seekg(0, ios::beg);
   do {
      read(reinterpret_cast<char *> (&reg),
sizeof(Registro));
      cout << reg.Nombre() << endl;
      if(gcount() > 0 && reg.Valido())
         ftemp.write(reinterpret_cast<char *> (&reg),
sizeof(Registro));
   } while (gcount() > 0);
   ftemp.close();
   close();
   remove("alea.bak");
   rename("alea.dat", "alea.bak");
   rename("alea.tmp", "alea.dat");
   open("alea.dat", ios::in | ios::out | ios::binary);
}

int main()
{
   Registro reg;
   Datos datos;
   int opcion;
   long numero;

   do {
      opcion = Menu();
      switch(opcion) {
         case '1': // Añadir registro
            reg.Leer();
            datos.Guardar(reg);
```

```
            break;
         case '2': // Mostrar registro
            system("cls");
            cout << "Mostrar registro: ";
            numero = LeeNumero();
            if(datos.Recupera(numero, reg))
               reg.Mostrar();
            break;
         case '3': // Eliminar registro
            system("cls");
            cout <<  "Eliminar registro: ";
            numero = LeeNumero();
            datos.Borrar(numero);
            break;
         case '4': // Mostrar todo
            numero = 0;
            system("cls");
            cout << "Nombre
Datos" << endl;
            while(datos.Recupera(numero, reg))
reg.Listar(numero++);
            cout << "pulsa return";
            cin.get();
            break;
      }
   } while(opcion != '0');
   return 0;
}

// Muestra un menú con las opciones disponibles y captura
una opción del usuario
int Menu()
{
   char resp[20];

   do {
      system("cls");
      cout << "MENU PRINCIPAL" << endl;
      cout << "--------------" << endl << endl;
      cout << "1- Insertar registro" << endl;
      cout << "2- Mostrar registro" << endl;
      cout << "3- Eliminar registro" << endl;
      cout << "4- Mostrar todo" << endl;
      cout << "0- Salir" << endl;
      cin.getline(resp, 20);
   } while(resp[0] < '0' && resp[0] > '4');
   return resp[0];
}
```

```
// Lee un número suministrado por el usuario
long LeeNumero()
{
   char numero[6];

   fgets(numero, 6, stdin);
   return atoi(numero);
}
```

5 Ordenar ficheros secuenciales

A veces necesitaremos ordenar el contenido de un fichero secuencial, ya sea de longitud de registro variable o constante.

Debido a la naturaleza de estos archivos, en general no será posible usar los métodos de ordenamiento que usaríamos con tablas en memoria. En muchas ocasiones trabajaremos con archivos muy grandes, de modo que será imposible ordenarlos en memoria y después reconstruirlos en disco.

Algoritmo de mezcla natural

En cuanto a los ficheros secuenciales, el método más usado es el de mezcla natural. Es válido para ficheros de tamaño de registro variable.

Es un buen método para ordenar barajas de naipes, por ejemplo.

Cada pasada se compone de dos fases. En la primera se separa el fichero original en dos auxiliares, los elementos se dirigen a uno u otro fichero separando los tramos de registros que ya estén ordenados. En la segunda fase los dos ficheros auxiliares se mezclan de nuevo de modo que de cada dos tramos se obtiene siempre uno ordenado. El proceso se repite hasta que sólo obtenemos un tramo.

Por ejemplo, supongamos los siguientes valores en un fichero de acceso secuencial, que ordenaremos de menor a mayor:

```
3, 1, 2, 4, 6, 9, 5, 8, 10, 7
```

Separaremos todos los tramos ordenados de este fichero:

```
[3], [1, 2, 4, 6, 9], [5, 8, 10], [7]
```

La primera pasada separará los tramos alternándolos en dos ficheros auxiliares:

```
aux1: [3],  [5, 8, 10]
```

```
aux2: [1, 2, 4, 6, 9],  [7]
```

Ahora sigue una pasada de mezcla, mezclaremos un tramo de cada fichero auxiliar en un único tramo:

```
mezcla: [1, 2, 3, 4, 6, 9], [5, 7, 8, 10]
```

Ahora repetimos el proceso, separando los tramos en los ficheros auxiliares:

```
aux1: [1, 2, 3, 4, 6, 9]
```

```
aux2: [5, 7, 8, 10]
```

Y de mezclándolos de nuevo:

```
mezcla: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

El fichero ya está ordenado, para verificarlo contaremos los tramos obtenidos después de cada proceso de mezcla, el fichero estará desordenado si nos encontramos más de un tramo.

Ejemplo

```
// mezcla.c : Ordenamiento de archivos secuenciales
// Ordena ficheros de texto por orden alfabético de líneas
// Usando el algoritmo de mezcla natural
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void Mostrar(FILE *fich);
void Mezcla(FILE *fich);
void Separar(FILE *fich, FILE **aux);
int Mezclar(FILE *fich, FILE **aux);

int main()
{
   FILE *fichero;

   fichero = fopen("mezcla.txt", "r+");
   puts("Fichero desordenado\n");
   Mostrar(fichero);
   puts("Ordenando fichero\n");
   Mezcla(fichero);
   puts("Fichero ordenado\n");
   Mostrar(fichero);
   fclose(fichero);
   system("PAUSE");
   return 0;
}

// Muestra el contenido del fichero "fich"
void Mostrar(FILE *fich)
{
   char linea[128];

   rewind(fich);
   fgets(linea, 128, fich);
   while(!feof(fich)) {
      puts(linea);
```

```
      fgets(linea, 128, fich);
   }
}

// Algoritmo de mezcla:
void Mezcla(FILE *fich)
{
   int ordenado;
   FILE *aux[2];

   // Bucle que se repite hasta que el fichero esté
ordenado:
   do {
      // Crea los dos ficheros auxiliares para separar los
tramos:
      aux[0] = fopen("aux1.txt", "w+");
      aux[1] = fopen("aux2.txt", "w+");
      rewind(fich);
      Separar(fich, aux);
      rewind(aux[0]);
      rewind(aux[1]);
      rewind(fich);
      ordenado = Mezclar(fich, aux);
      fclose(aux[0]);
      fclose(aux[1]);
   } while(!ordenado);
   // Elimina los ficheros auxiliares:
   remove("aux1.txt");
   remove("aux2.txt");
}

// Separa los tramos ordenados alternando entre los ficheros
auxiliares:
void Separar(FILE *fich, FILE **aux)
{
   char linea[128], anterior[2][128];
   int salida = 0;

   // Volores iniciales para los últimos valores
   // almacenados en los ficheros auxiliares
   strcpy(anterior[0], "");
   strcpy(anterior[1], "");
   // Captura la primero línea:
   fgets(linea, 128, fich);
   while(!feof(fich)) {
      // Decide a qué fichero de salida corresponde la línea
leída:
      if(salida == 0 && strcmp(linea, anterior[0]) < 0)
```

```
salida = 1;
      else if(salida == 1 && strcmp(linea, anterior[1]) < 0)
salida = 0;
      // Almacena la línea actual como la última añadida:
      strcpy(anterior[salida], linea);
      // Añade la línea al fichero auxiliar:
      fputs(linea, aux[salida]);
      // Lee la siguiente línea:
      fgets(linea, 128, fich);
   }
}

// Mezcla los ficheros auxiliares:
int Mezclar(FILE *fich, FILE **aux)
{
   char ultima[128], linea[2][128], anterior[2][128];
   int entrada;
   int tramos = 0;

   // Lee la primera línea de cada fichero auxiliar:
   fgets(linea[0], 128, aux[0]);
   fgets(linea[1], 128, aux[1]);
   // Valores iniciales;
   strcpy(ultima, "");
   strcpy(anterior[0], "");
   strcpy(anterior[1], "");
   // Bucle, mientras no se acabe ninguno de los ficheros
auxiliares (quedan tramos por mezclar):
   while(!feof(aux[0]) && !feof(aux[1])) {
      // Selecciona la línea que se añadirá:
      if(strcmp(linea[0], linea[1]) <= 0) entrada = 0; else
entrada = 1;
      // Almacena el valor como el último añadido:
      strcpy(anterior[entrada], linea[entrada]);
      // Añade la línea al fichero:
      fputs(linea[entrada], fich);
      // Lee la siguiente línea del fichero auxiliar:
      fgets(linea[entrada], 128, aux[entrada]);
      // Verificar fin de tramo, si es así copiar el resto
del otro tramo:
      if(strcmp(anterior[entrada], linea[entrada]) > 0) {
         if(!entrada) entrada = 1; else entrada = 0;
         tramos++;
         // Copia lo que queda del tramo actual al fichero
de salida:
         do {
            strcpy(anterior[entrada], linea[entrada]);
            fputs(linea[entrada], fich);
```

```
            fgets(linea[entrada], 128, aux[entrada]);
         } while(!feof(aux[entrada]) &&
strcmp(anterior[entrada], linea[entrada]) <= 0);
      }
   }

   // Añadir tramos que queden sin mezclar:
   if(!feof(aux[0])) tramos++;
   while(!feof(aux[0])) {
      fputs(linea[0], fich);
      fgets(linea[0], 128, aux[0]);
   }
   if(!feof(aux[1])) tramos++;
   while(!feof(aux[1])) {
      fputs(linea[1], fich);
      fgets(linea[1], 128, aux[1]);
   }
   return(tramos == 1);
}
```

Ordenar archivos es siempre una tarea muy lenta y requiere mucho tiempo. Este algoritmo, además requiere el doble de espacio en disco del que ocupa el fichero a ordenar, por ejemplo, para ordenar un fichero de 500 megas se necesitan otros 500 megas de disco libres.

Sin embargo, un fichero como el mencionado, sería muy difícil de ordenar en memoria.

6 Ordenar ficheros de acceso aleatorio

Cuando trabajemos con ficheros de acceso secuencial con tamaño de registro constante, podremos aplicar los mismos algoritmos de ordenación que con tablas en memoria, ya que es posible acceder a cada registro para lectura y escritura.

En el caso de ficheros de acceso aleatorio con tamaño de registro variable, los trataremos como si fueran secuenciales.

Algoritmo Quicksort

Por supuesto, hay que elegir un algoritmo que impleque un mínimo de lecturas y escrituras en el fichero, y preferentemente, que éstas operaciones estén los más próximas posible entre si. Resulta muy costoso, en términos de tiempo de ejecución, hacer muchas lecturas y escrituras en disco, y más si los puntos donde se realizan están muy separados entre ellos.

Como ejemplo, usaremos el algoritmo de ordenación quicksort, adaptándolo para ordenar ficheros.

Usaremos el programa de ejemplo que usamos para los archivos de acceso aleatorio "alea.cpp". Y añadiremos una nueva opción para ordenar el archivo.

Ejemplo

```
// alea2.c: Ejemplo de ficheros de acceso aleatorio.
// Incluye la opción de ordenar el archivo.
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

struct stRegistro {
   char valido;  // Campo que indica si el registro es
valido S->Válido, N->Inválido
   char nombre[34];
   int dato[4];
};

int Menu();
void Leer(struct stRegistro *reg);
void Mostrar(struct stRegistro *reg);
void Listar(long n, struct stRegistro *reg);
long LeeNumero();
void Empaquetar(FILE **fa);
void Ordenar(FILE *fa);
void Intercambia(FILE *fa, long iz, long de);
char *LeeCampo(FILE *fa, long n, char *buf);
void QuickSort(FILE *fa, long inicio, long final);

int main()
{
   struct stRegistro reg;
   FILE *fa;
   int opcion;
   long numero;
   fa = fopen("alea.dat", "r+b");          // Este modo
permite leer y escribir
   if(!fa) fa = fopen("alea.dat", "w+b");  // si el fichero
no existe, lo crea.
   do {
      opcion = Menu();
      switch(opcion) {
         case '1': // Añadir registro
            Leer(&reg);
            // Insertar al final:
            fseek(fa, 0, SEEK_END);
            fwrite(&reg, sizeof(struct stRegistro), 1, fa);
            break;
         case '2': // Mostrar registro
            system("cls");
            printf("Mostrar registro: ");
            numero = LeeNumero();
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
            fread(&reg, sizeof(struct stRegistro), 1, fa);
            Mostrar(&reg);
            break;
```

```
         case '3': // Eliminar registro
            system("cls");
            printf("Eliminar registro: ");
            numero = LeeNumero();
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
            fread(&reg, sizeof(struct stRegistro), 1, fa);
            reg.valido = 'N';
            fseek(fa, numero*sizeof(struct stRegistro),
SEEK_SET);
            fwrite(&reg, sizeof(struct stRegistro), 1, fa);
            break;
         case '4': // Mostrar todo
            rewind(fa);
            numero = 0;
            system("cls");
            printf("Nombre
Datos\n");
            while(fread(&reg, sizeof(struct stRegistro), 1,
fa))
               Listar(numero++, &reg);
            system("PAUSE");
            break;
         case '5': // Eliminar marcados
            Empaquetar(&fa);
            break;
         case '6': // Ordenar
            Empaquetar(&fa);
            Ordenar(fa);
            break;
      }
   } while(opcion != '0');
   fclose(fa);
   return 0;
}

// Muestra un menú con las opciones disponibles y captura
una opción del usuario
int Menu()
{
   char resp[20];
   do {
      system("cls");
      printf("MENU PRINCIPAL\n");
      printf("--------------\n\n");
      printf("1- Insertar registro\n");
      printf("2- Mostrar registro\n");
      printf("3- Eliminar registro\n");
```

```
      printf("4- Mostrar todo\n");
      printf("5- Eliminar registros marcados\n");
      printf("6- Ordenar fichero\n");
      printf("0- Salir\n");
      fgets(resp, 20, stdin);
   } while(resp[0] < '0' && resp[0] > '6');
   return resp[0];
}

// Permite que el usuario introduzca un registro por
pantalla
void Leer(struct stRegistro *reg)
{
   int i;
   char numero[6];
   system("cls");
   printf("Leer registro:\n\n");
   reg->valido = 'S';
   printf("Nombre: ");
   fgets(reg->nombre, 34, stdin);
   // la función fgets captura el retorno de línea, hay que
eliminarlo:
   for(i = strlen(reg->nombre)-1; i && reg->nombre[i] < ' ';
i--)
      reg->nombre[i] = 0;
   for(i = 0; i < 4; i++) {
      printf("Dato[%1d]: ", i);
      fgets(numero, 6, stdin);
      reg->dato[i] = atoi(numero);
   }
}

// Muestra un registro en pantalla, si no está marcado como
borrado
void Mostrar(struct stRegistro *reg)
{
   int i;
   system("cls");
   if(reg->valido == 'S') {
      printf("Nombre: %s\n", reg->nombre);
      for(i = 0; i < 4; i++)
         printf("Dato[%1d]: %d\n", i, reg->dato[i]);
   }
   system("PAUSE");
}

// Muestra un registro por pantalla en forma de listado,
// si no está marcado como borrado
```

```
void Listar(long n, struct stRegistro *reg)
{
   int i;
   if(reg->valido == 'S') {
      printf("[%6ld] %-34s", n, reg->nombre);
      for(i = 0; i < 4; i++) printf(", %4d", reg->dato[i]);
      printf("\n");
   }
}

// Lee un número suministrado por el usuario
long LeeNumero()
{
   char numero[6];
   fgets(numero, 6, stdin);
   return atoi(numero);
}

// Elimina los registros marcados como borrados
void Empaquetar(FILE **fa)
{
   FILE *ftemp;
   struct stRegistro reg;

   ftemp = fopen("alea.tmp", "wb");
   rewind(*fa);
   while(fread(&reg, sizeof(struct stRegistro), 1, *fa))
      if(reg.valido == 'S')
         fwrite(&reg, sizeof(struct stRegistro), 1, ftemp);
   fclose(ftemp);
   fclose(*fa);
   remove("alea.bak");
   rename("alea.dat", "alea.bak");
   rename("alea.tmp", "alea.dat");
   *fa = fopen("alea.dat", "r+b");
}

void Ordenar(FILE *fa)
{
   long nRegs;
   fseek(fa, 0, SEEK_END);
   nRegs = ftell(fa)/sizeof(struct stRegistro);
   QuickSort(fa, 0L, nRegs-1);
}

void QuickSort(FILE *fa, long inicio, long final)
{
   long iz, de;
```

```
   char mitad[34];
   static char cad[34];

   iz = inicio;
   de = final;
   strcpy(mitad, LeeCampo(fa, (iz+de)/2, cad));
   do {
      while(strcmp(LeeCampo(fa, iz, cad), mitad) < 0 && iz <
final) iz++;
      while(strcmp(mitad, LeeCampo(fa, de, cad)) < 0 && de >
inicio) de--;
      if(iz < de) Intercambia(fa, iz, de);
      if(iz <= de) {
         iz++;
         de--;
      }
   } while(iz <= de);
   if(inicio < de) QuickSort(fa, inicio, de);
   if(iz < final) QuickSort(fa, iz, final);
}

char *LeeCampo(FILE *fa, long n, char *buf)
{
   struct stRegistro reg;
   fseek(fa, n*sizeof(struct stRegistro), SEEK_SET);
   fread(&reg, sizeof(struct stRegistro), 1, fa);
   strcpy(buf, reg.nombre);
   return buf;
}

void Intercambia(FILE *fa, long iz, long de)
{
   struct stRegistro reg1, reg2;
   fseek(fa, iz*sizeof(struct stRegistro), SEEK_SET);
   fread(&reg1, sizeof(struct stRegistro), 1, fa);
   fseek(fa, de*sizeof(struct stRegistro), SEEK_SET);
   fread(&reg2, sizeof(struct stRegistro), 1, fa);
   fseek(fa, iz*sizeof(struct stRegistro), SEEK_SET);
   fwrite(&reg2, sizeof(struct stRegistro), 1, fa);
   fseek(fa, de*sizeof(struct stRegistro), SEEK_SET);
   fwrite(&reg1, sizeof(struct stRegistro), 1, fa);
}
```

El algoritmo que hemos usado es bastante bueno para ordenar ficheros, ya que requiere muy pocos intercambios de registros, pero de todos modos, con ficheros grandes puede ser un proceso muy

lento. En general es preferible no ordenar los ficheros, salvo que sea muy necesario.

Veamos ahora un ejemplo basado en streams para C++:

```
// alea2.cpp: Ejemplo de ficheros de acceso aleatorio.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

using namespace std;

// Funciones auxiliares:
int Menu();
long LeeNumero();

// Clase registro.
class Registro {
  public:
   Registro(char *n=NULL, int d1=0, int d2=0, int d3=0, int
d4=0) : valido('S') {
     if(n) strcpy(nombre, n); else strcpy(nombre, "");
     dato[0] = d1;
     dato[1] = d2;
     dato[2] = d3;
     dato[3] = d4;
   }
   void Leer();
   void Mostrar();
   void Listar(long n);

   const bool Valido() { return valido == 'S'; }
   const char *Nombre() { return nombre; }
  private:
   char valido;  // Campo que indica si el registro es
válido
                 // S->Válido, N->Inválido
   char nombre[34];
   int dato[4];
};

// Implementaciones de clase Registro:
// Permite que el usuario introduzca un registro por
pantalla
```

```
void Registro::Leer() {
   system("cls");
   cout << "Leer registro:" << endl << endl;
   valido = 'S';
   cout << "Nombre: ";
   cin.getline(nombre, 34);
   for(int i = 0; i < 4; i++) {
      cout << "Dato[" << i << "]: ";
      dato[i] = LeeNumero();
   }
}

// Muestra un registro en pantalla, si no está marcado como
borrado
void Registro::Mostrar()
{
   system("cls");
   if(Valido()) {
      cout << "Nombre: " << nombre << endl;
      for(int i = 0; i < 4; i++)
         cout << "Dato[" << i << "]: " << dato[i] << endl;
   }
   cout << "Pulsa una tecla";
   cin.get();
}

// Muestra un registro por pantalla en forma de listado,
// si no está marcado como borrado
void Registro::Listar(long n) {
   int i;
   if(Valido()) {
      cout << "[" << setw(6) << n << "] ";
      cout << setw(34) << nombre;
      for(i = 0; i < 4; i++)
         cout << ", " << setw(4) << dato[i];
      cout << endl;
   }
}

// Clase Datos, almacena y trata los datos.
class Datos :public fstream {
  public:
   Datos() : fstream("alea.dat", ios::in | ios::out |
ios::binary) {
      if(!good()) {
         open("alea.dat", ios::in | ios::out | ios::trunc |
ios::binary);
         cout << "fichero creado" << endl;
```

```
          cin.get();
       }
    }
   ~Datos() {
      Empaquetar();
    }
    void Guardar(Registro &reg);
    bool Recupera(long n, Registro &reg);
    void Borrar(long n);
    void Ordenar();

  private:
    void Empaquetar();
    void Intercambia(long iz, long de);
    char *LeeCampo(long n, char *buf);
    void QuickSort(long inicio, long final);
};

// Implementación de la clase Datos.
void Datos::Guardar(Registro &reg) {
   // Insertar al final:
   clear();
   seekg(0, ios::end);
   write(reinterpret_cast<char *> (&reg), sizeof(Registro));
   cout << reg.Nombre() << endl;
}

bool Datos::Recupera(long n, Registro &reg) {
   clear();
   seekg(n*sizeof(Registro), ios::beg);
   read(reinterpret_cast<char *> (&reg), sizeof(Registro));
   return gcount() > 0;
}

// Marca el registro como borrado:
void Datos::Borrar(long n) {
   char marca;

   clear();
   marca = 'N';
   seekg(n*sizeof(Registro), ios::beg);
   write(&marca, 1);
}

// Elimina los registros marcados como borrados
void Datos::Empaquetar() {
   ofstream ftemp("alea.tmp", ios::out);
   Registro reg;
```

```
   clear();
   seekg(0, ios::beg);
   do {
      read(reinterpret_cast<char *> (&reg),
sizeof(Registro));
      if(gcount() > 0 && reg.Valido())
         ftemp.write(reinterpret_cast<char *> (&reg),
sizeof(Registro));
   } while (gcount() > 0);
   ftemp.close();
   close();
   remove("alea.bak");
   rename("alea.dat", "alea.bak");
   rename("alea.tmp", "alea.dat");
   open("alea.dat", ios::in | ios::out | ios::binary);
}

// Ordenar el fichero:
void Datos::Ordenar()
{
   long nRegs;

   Empaquetar();

   clear();
   seekg(0, ios::end);
   nRegs = tellg()/sizeof(Registro);
   QuickSort(0, nRegs-1);
}

// Funciones auxiliares para ordenar
void Datos::QuickSort(long inicio, long final)
{
   long iz, de;
   char mitad[34];
   static char cad[34];

   iz = inicio;
   de = final;
   strcpy(mitad, LeeCampo((iz+de)/2, cad));
   do {
      while(strcmp(LeeCampo(iz, cad), mitad) < 0 && iz <
final) iz++;
      while(strcmp(mitad, LeeCampo(de, cad)) < 0 && de >
inicio) de--;
      if(iz < de) Intercambia(iz, de);
      if(iz <= de) {
```

```
         iz++;
         de--;
      }
   } while(iz <= de);
   if(inicio < de) QuickSort(inicio, de);
   if(iz < final) QuickSort(iz, final);
}

char* Datos::LeeCampo(long n, char *buf)
{
   Registro reg;

   seekg(n*sizeof(Registro), ios::beg);
   read(reinterpret_cast<char *> (&reg), sizeof(Registro));
   strcpy(buf, reg.Nombre());
   return buf;
}

void Datos::Intercambia(long iz, long de)
{
   Registro reg1, reg2;

   seekg(iz*sizeof(Registro), ios::beg);
   read(reinterpret_cast<char *> (&reg1), sizeof(Registro));
   seekg(de*sizeof(Registro), ios::beg);
   read(reinterpret_cast<char *> (&reg2), sizeof(Registro));
   seekp(iz*sizeof(Registro), ios::beg);
   write(reinterpret_cast<char *> (&reg2),
sizeof(Registro));
   seekp(de*sizeof(Registro), ios::beg);
   write(reinterpret_cast<char *> (&reg1),
sizeof(Registro));
}

int main()
{
   Registro reg;
   Datos datos;
   int opcion;
   long numero;
   do {
      opcion = Menu();
      switch(opcion) {
         case '1': // Añadir registro
            reg.Leer();
            datos.Guardar(reg);
            break;
         case '2': // Mostrar registro
```

```
            system("cls");
            cout << "Mostrar registro: ";
            numero = LeeNumero();
            if(datos.Recupera(numero, reg))
               reg.Mostrar();
            break;
         case '3': // Eliminar registro
            system("cls");
            cout <<  "Eliminar registro: ";
            numero = LeeNumero();
            datos.Borrar(numero);
            break;
         case '4': // Mostrar todo
            numero = 0;
            system("cls");
            cout << "Nombre
Datos" << endl;
            while(datos.Recupera(numero, reg))
reg.Listar(numero++);
            cout << "pulsa return";
            cin.get();
            break;
         case '5': // Ordenar
            datos.Ordenar();
            break;
      }
   } while(opcion != '0');
   return 0;
}

// Muestra un menú con las opciones disponibles y captura
una opción del usuario
int Menu()
{
   char resp[20];
   do {
      system("cls");
      cout << "MENU PRINCIPAL" << endl;
      cout << "--------------" << endl << endl;
      cout << "1- Insertar registro" << endl;
      cout << "2- Mostrar registro" << endl;
      cout << "3- Eliminar registro" << endl;
      cout << "4- Mostrar todo" << endl;
      cout << "5- Ordenar" << endl;
      cout << "0- Salir" << endl;
      cin.getline(resp, 20);
   } while(resp[0] < '0' && resp[0] > '5');
   return resp[0];
```

```
}

// Lee un número suministrado por el usuario
long LeeNumero()
{
   char numero[6];
   fgets(numero, 6, stdin);
   return atoi(numero);
}
```

7 Ficheros de índices

Mantener grandes ficheros de datos ordenados es muy costoso, ya que requiere mucho tiempo de procesador. Afortunadamente, existe una alternativa mucho mejor: indicarlos (o indexarlos).

Para indicar un archivo normalmente se suele generar un archivo auxiliar de índices. Existen varios métodos, de los que veremos algunos. El más sencillo es crear un archivo plano que sólo contenga registros con dos campos: el campo o la expresión por la que queremos ordenar el archivo, y un campo con un índexe que almecene la posición del registro indicado en el archivo de datos.

Por ejemplo, supongamos que tenemos un archivo de datos con la siguiente estructura de registro:

```
struct stRegistro {
   char nombre[32];
   char apellido[2][32];
   char telefono[12];
   char calle[45];
   int numero;
   char ciudad[32];
   char fechaNacimiento[9]; // formato AAAAMMDD: Año, mes y
día
   char estadoCivil;
   int hijos;
}
```

Imaginemos que necesitamos buscar un registro a partir del número de teléfono. Si no tenemos el archivo ordenado por ese campo, estaremos obligados a leer todos los registros del archivo hasta encontrar el que buscamos, y si el número no está, tendremos que leer todos los registros que existan.

Si tenemos el archivo ordenado por números de teléfono podremos aplicar un algoritmo de búsqueda. Pero si también queremos hacer búsquedas por otros campos, estaremos obligados a ordenar de nuevo el archivo.

La solución es crear un fichero de índices, cada registro de este archivo tendrá la siguiente estructura:

```
struct stIndiceTelefono {
   char telefono[12];
   long indice;
}
```

Crearemos el fichero de índices a partir del archivo de datos, asignando a cada registro el campo "telefono" y el número de registro correspondiente. Veamos un ejemplo:

```
000: [Fulanito] [Pérez] [Sanchez] [12345678] [Mayor] [15]
[Lisboa] [19540425] [S] [0]
001: [Fonforito] [Fernandez] [López] [84565456] [Baja] [54]
[Londres] [19750924] [C] [3]
002: [Tantolito] [Jimenez] [Fernandez] [45684565] [Alta]
[153] [Berlin] [19840628] [S] [0]
003: [Menganito] [Sanchez] [López] [23254532] [Diagonal]
[145] [Barcelona] [19650505] [C] [1]
004: [Tulanito] [Sanz] [Sanchez] [54556544] [Pez] [18]
[Dublín] [19750111] [S] [0]
```

Generamos un fichero de índices:

```
[12345678][000]
[84565456][001]
[45684565][002]
[23254532][003]
[54556544][004]
```

Y lo ordenamos:

```
[12345678][000]
[23254532][003]
[45684565][002]
[54556544][004]
[84565456][001]
```

Ahora, cuando queramos buscar un número de teléfono, lo haremos en el fichero de índices, por ejemplo el "54556544" será el registro número 3, y le corresponde el índice "004". Con ese índice podemos acceder directamente al archivo de datos, y veremos que el número corresponde a "Tulanito Sanz Sanchez".

Por supuesto, nada nos impide tener más ficheros de índices, para otros campos.

El mayor problema es mantener los ficheros de índices ordenados a medida que añadimos, eliminamos o modificamos registros. Pero al ser los registros de índices más pequeños, los ficheros son más manejables, pudiendo incluso almacenarse en memoria en muchos casos.

Ejemplo

Veramos un ejemplo de implementación de índices:

```
// indices.cpp: Ejemplo de ficheros de acceso aleatorio con
índices.
#include <cstdio>
#include <cstdlib>
#include <cstring>

using namespace std;

struct stRegistro {
   char valido;  // Campo que indica si el registro es
valido S->Válido, N->Inválido
   char nombre[34];
   char apellido[2][34];
   char telefono[10];
```

```
};

struct stIndice {
   char telefono[10];
   long indice;
};

int Menu();
void Capturar(stRegistro &reg);
void EliminarRetornoLinea(char *cad);
void Leer(FILE *fa, stRegistro &reg, char *telefono);
void Insertar(FILE *fa, stRegistro &reg);
void Mostrar(stRegistro &reg);
void ListarPorTelefonos(FILE *fa);
void ListarNatural(FILE *fa);
void ReconstruirIndices(FILE *fa);
// Funciones para ordenar el fichero de índices:
void Intercambia(FILE *fa, long iz, long de);
char *LeeCampo(FILE *fa, long n, char *buf);
void QuickSort(FILE *fa, long inicio, long final);

int main()
{
   stRegistro reg;
   FILE *fa;
   int opcion;
   char telefono[10];

   fa = fopen("indices.dat", "r+b");          // Este modo
permite leer y escribir
   if(!fa) fa = fopen("indices.dat", "w+b");  // si el
fichero no existe, lo crea.

   do {
      opcion = Menu();
      switch(opcion) {
         case '1': // Insertar registro
            Capturar(reg);
            Insertar(fa, reg);
            break;
         case '2': // Buscar registro
            system("cls");
            printf("Buscar registro: ");
            do {
               fgets(telefono, 10, stdin);
               EliminarRetornoLinea(telefono);
            } while(strlen(telefono) < 1);
            Leer(fa, reg, telefono);
```

```
            Mostrar(reg);
            break;
         case '3': // Indicar archivo
            system("cls");
            printf("Indicando archivo: ");
            ReconstruirIndices(fa);
            break;
         case '4': // Mostrar todo por orden de teléfonos
            ListarPorTelefonos(fa);
            break;
         case '5': // Mostrar todo por orden natural
            ListarNatural(fa);
            break;
      }
   } while(opcion != '0');
   fclose(fa);
   return 0;
}

// Muestra un menú con las opciones disponibles y captura
una opción del usuario
int Menu()
{
   char resp[20];

   do {
      system("cls");
      printf("MENU PRINCIPAL\n");
      printf("--------------\n\n");
      printf("1- Insertar registro\n");
      printf("2- Buscar registro\n");
      printf("3- Reindicar archivo\n");
      printf("4- Listar por orden de teléfonos\n");
      printf("5- Listar por orden natural\n");
      printf("0- Salir\n");
      fgets(resp, 20, stdin);
   } while(resp[0] < '0' && resp[0] > '5');
   return resp[0];
}

// Permite que el usuario introduzca un registro por
pantalla
void Capturar(stRegistro &reg)
{
   int i;
   char numero[6];

   system("cls");
```

```
   printf("Leer registro:\n\n");
   reg.valido = 'S';
   printf("Nombre: ");
   fgets(reg.nombre, 34, stdin);
   EliminarRetornoLinea(reg.nombre);
   printf("Primer apellido: ");
   fgets(reg.apellido[0], 34, stdin);
   EliminarRetornoLinea(reg.apellido[0]);
   printf("Segundo apellido: ");
   fgets(reg.apellido[1], 34, stdin);
   EliminarRetornoLinea(reg.apellido[1]);
   printf("Teléfono: ");
   fgets(reg.telefono, 10, stdin);
   EliminarRetornoLinea(reg.telefono);
}

// Elimina los caracteres de retorno de línea al final de
cadena
void EliminarRetornoLinea(char *cad)
{
   int i;
   // la función fgets captura el retorno de línea, hay que
eliminarlo:
   for(i = strlen(cad)-1; i >= 0 && cad[i] < ' '; i--)
cad[i] = 0;
}

// Muestra un registro en pantalla, si no está marcado como
borrado
void Mostrar(stRegistro &reg)
{
   int i;

   if(reg.valido == 'S') {
      printf("Nombre: %s %s %s\n", reg.nombre,
reg.apellido[0], reg.apellido[1]);
      printf("Número de teléfono: %s\n", reg.telefono);
   }
   system("PAUSE");
}

// Lee el registro desde el fichero de datos con el teléfono
dado
void Leer(FILE *fa, stRegistro &reg, char *telefono)
{
   FILE *fi;
   stIndice ind;
   long inf, sup, n, nRegs;
```

```
   fi = fopen("indices.ind", "rb");
   fseek(fi, 0, SEEK_END);
   nRegs = ftell(fi)/sizeof(stIndice);
   // Búsqueda binaria:
   inf = 0;
   sup = nRegs-1;
   do {
      n = inf+(sup-inf)/2;
      fseek(fi, n*sizeof(stIndice), SEEK_SET);
      fread(&ind, sizeof(stIndice), 1, fi);
      if(strcmp(ind.telefono, telefono) < 0) inf = n+1;
      else sup = n-1;
   } while(inf <= sup && strcmp(ind.telefono, telefono));
   // Si se encontró el teléfono, lee el registro, si no
muestra mensaje.
   if(!strcmp(ind.telefono, telefono)) {
      fseek(fa, ind.indice*sizeof(stRegistro), SEEK_SET);
      fread(&reg, sizeof(stRegistro), 1, fa);
   }
   else {
      reg.valido = 'N';
      printf("Registro no encontrado\n");
   }
   fclose(fi);
}

// Añade un registro al archivo de datos y reconstruye los
índices
void Insertar(FILE *fa, stRegistro &reg)
{
   // Insertar al final:
   fseek(fa, 0, SEEK_END);
   fwrite(&reg, sizeof(stRegistro), 1, fa);
   ReconstruirIndices(fa);
}

// Lista todos los registros ordenados por el número de
teléfono
void ListarPorTelefonos(FILE *fa)
{
   FILE *fi;
   stIndice ind;
   stRegistro reg;

   system("cls");
   fi = fopen("indices.ind", "rb");
   while(fread(&ind, sizeof(stIndice), 1, fi)) {
```

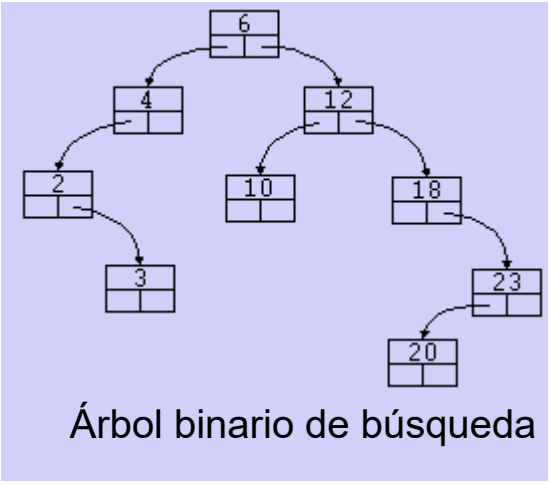
```
      fseek(fa, ind.indice*sizeof(stRegistro), SEEK_SET);
      fread(&reg, sizeof(stRegistro), 1, fa);
      printf("%s %s %s %s\n", reg.nombre, reg.apellido[0],
         reg.apellido[1], reg.telefono);
   }
   fclose(fi);
   system("PAUSE");
}

// Lista todos los registros del archivo de datos por el
orden en que se
// insertaron.
void ListarNatural(FILE *fa)
{
   stRegistro reg;

   rewind(fa);
   system("cls");
   while(fread(&reg, sizeof(stRegistro), 1, fa))
      printf("%s %s %s %s\n", reg.nombre, reg.apellido[0],
         reg.apellido[1], reg.telefono);
   system("PAUSE");
}

// Reconstruye el archivo de índices
void ReconstruirIndices(FILE *fa)
{
   long n=0;
   FILE *fi;
   stRegistro reg;
   stIndice ind;

   // Crea el fichero de índices a partir del archivo de
datos:
   fi = fopen("indices.ind", "w+b");
   rewind(fa);
   while(fread(&reg, sizeof(stRegistro), 1, fa)) {
      strcpy(ind.telefono, reg.telefono);
      ind.indice = n++;
      fwrite(&ind, sizeof(stIndice), 1, fi);
   }
   // Ordena usando el algoritmo Quicksort:
   QuickSort(fi, 0, n-1);
   fclose(fi);
}

// Implementación del algoritmo Quicksort para fichero de
índices
```

```
void QuickSort(FILE *fi, long inicio, long final)
{
   long iz, de;
   char mitad[10];
   static char cad[10];

   iz = inicio;
   de = final;
   strcpy(mitad, LeeCampo(fi, (iz+de)/2, cad));
   do {
      while(strcmp(LeeCampo(fi, iz, cad), mitad) < 0 && iz <
final) iz++;
      while(strcmp(mitad, LeeCampo(fi, de, cad)) < 0 && de >
inicio) de--;
      if(iz < de) Intercambia(fi, iz, de);
      if(iz <= de) {
         iz++;
         de--;
      }
   } while(iz <= de);
   if(inicio < de) QuickSort(fi, inicio, de);
   if(iz < final) QuickSort(fi, iz, final);
}

char *LeeCampo(FILE *fi, long n, char *buf)
{
   stIndice ind;

   fseek(fi, n*sizeof(stIndice), SEEK_SET);
   fread(&ind, sizeof(stIndice), 1, fi);
   strcpy(buf, ind.telefono);
   return buf;
}

void Intercambia(FILE *fi, long iz, long de)
{
   stIndice reg1, reg2;

   fseek(fi, iz*sizeof(stIndice), SEEK_SET);
   fread(&reg1, sizeof(stIndice), 1, fi);
   fseek(fi, de*sizeof(stIndice), SEEK_SET);
   fread(&reg2, sizeof(stIndice), 1, fi);
   fseek(fi, iz*sizeof(stIndice), SEEK_SET);
   fwrite(&reg2, sizeof(stIndice), 1, fi);
   fseek(fi, de*sizeof(stIndice), SEEK_SET);
   fwrite(&reg1, sizeof(stIndice), 1, fi);
}
```

Aún no hemos llegado al mayor nivel de optimización, nuestro último ejemplo requiere reconstruir el fichero de índices cada vez que se añade o se elimina un registro.

8 Ficheros indicados no ordenados: árboles binarios

Para evitar tener que reconstruir el fichero de índices cada vez que se actualiza el archivo de datos existen varios métodos. Veremos ahora cómo implementar árboles binarios.



Árbol binario de búsqueda

Para ello construiremos una estructura en árbol mediante una tabla almacenada en un archivo de disco.

La estructura para cada nodo del árbol es una extensión de la que usamos en el capítulo anterior, tan sólo añadiremos dos campos extra para apuntar a otros nodos:

```
struct stNodo {
   char telefono[10];
   long indice;
   long menor, mayor;
};
```

Crearemos el fichero de índices estructurados en árbol a partir del archivo de datos, asignando a cada registro el campo "telefono" y el número de registro correspondiente y añadiendo los enlaces a otros nodos. Veamos un ejemplo:

```
000: [Fulanito] [Pérez] [Sanchez] [12345678] [Mayor] [15]
[Lisboa] [19540425] [S] [0]
001: [Fonforito] [Fernandez] [López] [84565456] [Baja] [54]
```

```
[Londres] [19750924] [C] [3]
002: [Tantolito] [Jimenez] [Fernandez] [45684565] [Alta]
[153] [Berlin] [19840628] [S] [0]
003: [Menganito] [Sanchez] [López] [23254532] [Diagonal]
[145] [Barcelona] [19650505] [C] [1]
004: [Tulanito] [Sanz] [Sanchez] [54556544] [Pez] [18]
[Dublín] [19750111] [S] [0]
```

Veremos cómo se actualiza el fichero de índices a medida que insertamos registros en el archivo de datos:

Paso uno:

```
[12345678][000][---][---]
```

Paso dos:

```
[12345678][000][---][001] <--
[84565456][001][---][---]
```

Paso tres:

```
[12345678][000][---][001]
[84565456][001][002][---] <--
[45684565][002][---][---]
```

Paso cuatro:

```
[12345678][000][---][001]
[84565456][001][002][---]
[45684565][002][003][---] <--
[23254532][003][---][---]
```

Paso cinco:

```
[12345678][000][---][001]
[84565456][001][002][---]
[45684565][002][003][004] <--
[23254532][003][---][---]
[54556544][004][---][---]
```

Como puede observarse, cada vez que se inserta un registro de datos, tan sólo hay que insertar un registro de índice y modificar otro.

Eliminar registros

Supongamos que queremos eliminar un registro de datos. En el archivo de datos simplemente lo marcamos como borrado. En teoría, mientras el registro no se elimine físicamente, no será necesario eliminar el registro de índice asociado. Simplemente estará apuntando a un registro marcado como borrado. Posteriormente, cuando purguemos el archivo de datos será necesario reconstruir el fichero de índices.

Duplicación de claves

No hay inconveniente en almacenar registros con claves duplicadas, tan sólo habrá que tener en cuenta que tendremos que almacenar un nodo para cada uno de ellos. Tomaremos un criterio para el árbol, la rama 'menor', y pasará a ser la rama 'menor o igual'.

Ventajas y desventajas

Este método tiene la ventaja de que no es necesario ordenar el archivo de índices, pero puede producir resultados mediocres o francamente malos. Por ejemplo, si los registros se introducen

ordenados, buscar por la clave del último registro insertado requerirá leer todos los nodos del árbol.

Para evitar eso se recurre a otros tipos de estructuras, como veremos en próximos capítulos.

Tabla de contenido