ARCHIVOS

1 Introducción

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. A continuación 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 encontraba la que estabamos 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 contante. En C usaremos estructuras para definir los registros. C dispone de funciones de librería 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.

2 Funciones y tipos C estándar:

Tipo FILE:

C define la estructura de datos FILE en el fichero de cabecesa "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 ANSI C

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

Abre un fichero cuyo nombre es la cadena apuntada por **nombre**, y adjudica un stream a ello. El argumento **modo** apunta a una cadena empezando con una serie de caracteres de la siguiente secuencia:

r	Abre un fichero de texto para lectura
w	Trunca, a longitud cero o crea un fichero de texto para escribir
a	Añade; abre o crea un fichero de texto para escribir al final del fichero (EOF)
rb	Abre un fichero en modo binario para lectura
wb	Trunca, a longitud cero o crea un fichero en modo binario para escribir
ab	Añade; abre o crea un fichero en modo binario para escribir al final del fichero (EOF)
r+	Abre un fichero de texto para actualización (lectura y escritura)
w+	Trunca, a longitud cero o crea un fichero de texto para actualización
a+	Añade; abre o crea un fichero de texto para actualización, escribiendo al final del fichero (EOF)
r+b ó **rb+**	Abre un fichero en modo binario para actualización (lectura y escritura)
w+b ó **wb+**	Trunca, a longitud cero o crea un fichero en modo binario para actualización
a+b ó **ab+**	Añade; abre o crea un fichero en modo binario para actualización, escribiendo al final del fichero (EOF)

Abriendo un fichero con el modo de lectura ('r' como el primer carácter del argumento **modo**) fallará si el fichero no existe o no puede ser leído. Abriendo el fichero con el modo de añadidura ('a' como el primer carácter del argumento **modo**) ocasiona todas las escrituras posteriores al fichero a ser forzadas al final de fichero (EOF) actual, sin considerar llamadas interventivas a la función fseek. En algunas implementaciones, abriendo un fichero en modo binario con el modo de añadidura ('b'como el segundo o

tercer carácter del argumento **modo**) puede colocar, inicialmente, el indicador de posición de ficheros para el stream más allá de los últimos datos escritos, debido al relleno de caracteres nulos.

Cuando un fichero es abierto con el modo de actualización ('+' como el segundo o tercer carácter del argumento **modo**), la entrada y salida pueden ser manipuladas en el stream asociado. Sin embargo, la salida no puede estar seguida directamente de una entrada sin tener una llamada interventiva a la función fflush o a una función de manipulación del fichero de posición (fseek, fsetpos, o rewind), y la entrada no puede estar seguida directamente de una salida sin tener una llamada interventiva a una función de manipulación del fichero de posición, al menos que el proceso de entrada encuentre un final de fichero (EOF). Abriendo (o creando) un fichero de texto con el modo de actualización puede en su lugar abrir (o crear) un stream binario en algunas implementaciones.

Cuando es abierto, un stream es almacenado completamente si, y sólo si, puede ser determinado que no hace referencia a un dispositivo interactivo. Los indicadores de error y final de fichero (EOF) para el stream son borrados.

Valor de retorno:

La función *fopen* retorna un puntero al objeto controlando el stream. Si el proceso de abertura no es realizado acabo, entonces retorna un puntero nulo.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[10] = "datos.dat";

   fichero = fopen( nombre, "w" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   if( !fclose(fichero) )
      printf( "Fichero cerrado\n" );
   else
```

```
    {
       printf( "Error: fichero NO CERRADO\n" );
       return 1;
    }

    return 0;
}
```

Función fclose ANSI C

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

El stream apuntado por **stream** será despejado y el fichero asociado, cerrado. Cualquier dato almacenado aún sin escribir para el stream será enviado al entorno local para ser escritos al fichero; cualquier dato almacenado aún sin leer será descartado. El stream es desasociado del fichero. Si el almacenamiento asociado fue automáticamente adjudicado, será desadjudicado.

Valor de retorno:

La función *fclose* retorna cero si el stream fue cerrado con éxito. Si se detectaron errores, entonces retorna EOF.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[10] = "datos.dat";

   fichero = fopen( nombre, "w" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }
```

```
    if( !fclose(fichero) )
       printf( "Fichero cerrado\n" );
    else
    {
       printf( "Error: fichero NO CERRADO\n" );
       return 1;
    }

    return 0;
}
```

Función fgetc ANSI C

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

Esta función obtiene el carácter siguiente (si está presente) como un **unsigned char** convertido a **int**, desde el stream de entrada apuntado por **stream**, y avanza el indicador de posición de ficheros asociado al stream (si está definido).

Valor de retorno:

La función *fgetc* retorna el carácter siguiente desde el stream de entrada apuntado por **stream**. Si el stream está en el final de fichero, el indicador del final de fichero para el stream es activado y *fgetc* retorna EOF. Si ocurre un error de lectura, el indicador de error para el stream es activado y *fgetc* retorna EOF.

Ejemplo:

```
#include <stdio.h>

int main()
{
    char nombre[10]="datos.dat";
    FILE *fichero;
    int i;

    fichero = fopen( nombre, "r" );
    printf( "Fichero: %s -> ", nombre );
    if( fichero )
```

```
      printf( "existe (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }
printf( "Los 18 primeros caracteres del fichero:
%s\n\n", nombre );
   for( i=1; i<=18; i++)   printf( "%c",
fgetc(fichero) );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
   }

   return 0;
}
```

Función fputc ANSI C

```
int fputc(int c, FILE *stream);
```

Esta función escribe el carácter indicado por **c** (convertido a un **unsigned char**) al stream de salida apuntado por **stream**, en la posición indicada por el indicador de posición de ficheros asociado al stream (si está definido), y avanza el indicador apropiadamente. Si el fichero no soporta peticiones de posición, o si el stream fue abierto con el modo de añado, el carácter es añadido al stream de salida.

Valor de retorno:

La función *fputc* retorna el carácter escrito. Si ocurre un error de escritura, el indicador de error para el stream es activado y *fputc* retorna EOF.

Ejemplo:

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

```
int main()
{
   char nombre[10]="datos.dat";
   FILE *fichero;
   int i;

   fichero = fopen( nombre, "a" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "existe o ha sido creado
(ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }
printf( "Escribimos las 18 primeras letras del
abecedario ingles en el fichero: %s\n\n", nombre
);
   for( i=0; i<18; i++)   printf( "%c",
fputc('a'+i, fichero) );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
   }

   return 0;
}
```

Función feof ANSI C

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

La función *feof* comprueba el indicador de final de fichero para el stream apuntado por **stream**.

Valor de retorno:

La función *feof* retorna un valor distinto a cero si y sólo si el indicador de final de fichero está activado para **stream**.

Ejemplo:

```
#include <stdio.h>

int main( void )
{
   int tamanyo=0;
   FILE *ficheroEntrada, *ficheroSaliada;
char nombreEntrada[11]="datos2.dat",
nombreSalida[11]="datos3.dat";

   ficheroEntrada = fopen( nombreEntrada, "r" );
   printf( "Fichero de Lectura: %s -> ",
nombreEntrada );
   if( ficheroEntrada )
      printf( "existe (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }
   ficheroSalida = fopen( nombreSalida, "w" );
   printf( "Fichero de Lectura: %s -> ",
nombreSalida );
   if( ficheroSalida )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   while( !feof(ficheroEntrada) )
   {
```

```
      fputc( fgetc(ficheroEntrada)+3,
ficheroSalida );    /* Desplazar cada carácter 3
caracteres: a -> d, b -> e, ... */
      tamanyo++;
   }
   printf( "El fichero \'%s\' contiene %d
caracteres.\n", nombreEntrada, tamanyo );

   if( !fclose(ficheroSalida) )
      printf( "Fichero: %s cerrado\n",
nombreSalida );
   else
   {
      printf( "Error: fichero: %s NO CERRADO\n",
nombreSalida );
      return 1;
   }
   if( !fclose(ficheroEntrada) )
      printf( "Fichero: %s cerrado\n",
nombreEntrada );
   else
   {
      printf( "Error: fichero: %s NO CERRADO\n",
nombreEntrada );
      return 1;
   }

   return 0;
```

Función rewind ANSI C

```
void rewind(FILE *stream);
```

La función *rewind* coloca el indicador de posición de fichero para el stream apuntado por **stream** al comienzo del fichero. Es equivalente a **(void) fseek(stream, 0L, SEEK_SET)** excepto que el indicador de errores para el stream es despejado.

Valor de retorno:

La función *rewind* no retorna ningún valor.

Ejemplo:

```
#include <stdio.h>

int main()
{
  char nombre[11] = "datos4.dat",
       mensaje[81]="Esto es nua rpueba usando
fgetpos y fsetpos.";
  FILE *fichero;
  fpos_t posicion=0, comienzo;

  fichero = fopen( nombre, "w+" );
  printf( "Fichero: %s -> ", nombre );
  if( fichero )
    printf( "creado (ABIERTO)\n" );
  else
  {
    printf( "Error (NO ABIERTO)\n" );
    return 1;
  }

  fgetpos( fichero, &comienzo );
  printf( "Posicion del fichero: %d\n", posicion
);

  fprintf( fichero, mensaje );
  printf( "\nEscrito: \"%s\"\n", mensaje );

  fgetpos( fichero, &posicion );
  printf( "Posicion del fichero: %d\n", posicion
);

  fsetpos( fichero, &comienzo );
  fprintf( fichero, "%s", "Esto es una prueba" );
  printf( "Corregiendo errores...Escrito: \"Esto
es una prueba\"\n" );

  fgetpos( fichero, &posicion );
  printf( "Posicion del fichero: %d\n", posicion
);

  rewind( fichero );
```

```
  printf( "\"Rebobinando\" el fichero -> Vuelta al
comienzo\n" );
  fgetpos( fichero, &posicion );
  printf( "Posicion del fichero: %d\n", posicion
);

  printf( "\nLeyendo del fichero \"%s\"\n", nombre
);
  fgets( mensaje, 81, fichero );
  printf( "\"%s\"\n\n", mensaje );

  fgetpos( fichero, &posicion );
  printf( "Posicion del fichero: %d\n", posicion
);

  if( !fclose(fichero) )
    printf( "Fichero cerrado\n" );
  else
  {
    printf( "Error: fichero NO CERRADO\n" );
    return 1;
  }

  return 0;
}
```

Función fgets ANSI C

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

Esta función lee como máximo uno menos que el número de caracteres indicado por **n** desde el stream apuntado por **stream** al array apuntado por **cadena**. Ningún carácter adicional es leído después del carácter de nueva línea (el cual es retenido) o después de un final de fichero (EOF). Un carácter nulo es escrito inmediatamente después del último carácter leído en el array.

Valor de retorno:

La función *fgets* retorna **cadena** si es realizada con éxito. Si un final de fichero (EOF) es encontrado y ningún carácter ha sido leído en el array, entonces el contenido del array permanece invariable y un puntero nulo es retornado. Si ocurre un error de lectura

durante el proceso, el contenido del array es indeterminado y un puntero nulo es retornado.

Ejemplo:

```
#include <stdio.h>

int main()
{
   char nombre[10]="datos.dat", linea[81];
   FILE *fichero;

   fichero = fopen( nombre, "r" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "existe (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   printf( "La primera linea del fichero: %s\n\n",
nombre );
   printf( "%s\n", fgets(linea, 81, fichero) );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
   }

   return 0;
}
```

Función fputs ANSI C

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

Esta función escribe la cadena apuntada por **cadena** al stream apuntado por **stream**. El carácter nulo no es escrito.

Valor de retorno:

La función *fputs* retorna EOF si ocurre un error de escritura, si no, entonces retorna un valor no negativo.

Ejemplo:

```
#include <stdio.h>

int main()
{
   char nombre[11]="datos2.dat";
   FILE *fichero;

   fichero = fopen( nombre, "w" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   fputs( "Esto es un ejemplo usando \'fputs\'\n",
fichero );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
   }

   return 0;
}
```

Función fread ANSI C

```
size_t fread(void *puntero, size_t tamanyo, size_t
nmemb, FILE *stream);
```

La función *fread* recibe, en el array apuntado por **puntero**, hasta **nmemb** de elementos cuyo tamaño es especificado por **tamanyo**, desde el stream apuntado por **stream**. El indicador de posición de ficheros para el stream (si está definido) es avanzado por el número de caracteres leídos correctamente. Si existe un error, el valor resultante del indicador de posición de ficheros para el stream es indeterminado. Si un elemento es parcialmente leído, entonces su valor es indeterminado.

Valor de retorno:

La función *fread* retorna el número de caracteres leídos correctamente, el cual puede ser menor que **nmemb** si se encuentra un error de lectura o un final de fichero. Si **tamanyo** o **nmemb** es cero, *fread* retorna cero, y el contenido del array y el estado del stream permanecen invariados.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[11] = "datos5.dat";
   unsigned int dinero[10] = { 23, 12, 45, 345,
512, 345, 654, 287, 567, 124 };
   unsigned int leer[10], i;

   fichero = fopen( nombre, "w+" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   printf( "Escribiendo cantidades:\n\n" );
```

```
   for( i=0; i<10; i++ )
     printf( "%d\t", dinero[i] );

   fwrite( dinero, sizeof(unsigned int), 10,
fichero );

   printf( "\nLeyendo los datos del fichero
\"%s\":\n", nombre );
   rewind( fichero );
   fread( leer, sizeof(unsigned int), 10, fichero
);

   for( i=0; i<10; i++ )
     printf( "%d\t", leer[i] );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
   }

   return 0;
}
```

Función fwrite ANSI C

```
size_t fwrite(const void *puntero, size_t tamanyo,
size_t nmemb, FILE *stream);
```

La función *fwrite* envía, desde el array apuntado por **puntero**, hasta **nmemb** de elementos cuyo tamaño es especificado por **tamanyo**, al stream apuntado por **stream**. El indicador de posición de ficheros para el stream (si está definido) es avanzado por el número de caracteres escritos correctamente. Si existe un error, el valor resultante del indicador de posición de ficheros para el stream es indeterminado.

Valor de retorno:

La función *fwrite* retorna el número de caracteres escritos correctamente, el cual puede ser menor que **nmemb**, pero sólo si se produce un error de escritura.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[11] = "datos5.dat";
   unsigned int dinero[10] = ;
   unsigned int leer[10], i;

   fichero = fopen( nombre, "w+" );
   printf( "Fichero: %s -> ", nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   printf( "Escribiendo cantidades:\n\n" );

   for( i=0; i<10; i++ )
     printf( "%d\t", dinero[i] );

   fwrite( dinero, sizeof(unsigned int), 10,
fichero );

   printf( "\nLeyendo los datos del fichero
\"%s\":\n", nombre );
   rewind( fichero );
   fread( leer, sizeof(unsigned int), 10, fichero
);

   for( i=0; i<10; i++ )
     printf( "%d\t", leer[i] );

   if( !fclose(fichero) )
      printf( "\nFichero cerrado\n" );
   else
   {
      printf( "\nError: fichero NO CERRADO\n" );
      return 1;
```

```
    }

    return 0;
}
```

Función fprintf ANSI C

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

Esta función envía datos al stream apuntado por **stream**, bajo el control de la cadena apuntada por **formato** que especifica cómo los argumentos posteriores son convertidos para la salida. Si hay argumentos insuficientes para el formato, el comportamiento no está definido. Si el formato termina mientras quedan argumentos, los argumentos restantes son evaluados (como siempre) pero ignorados. La función retorna control cuando el final de la cadena de formato es encontrado.

Valor de retorno:

La función *fprintf* retorna el número de caracteres transmitidos, o un valor negativo si un error de salida se ha producido.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[10] = "datos.dat";
   unsigned int i;

   fichero = fopen( nombre, "w" );
   printf( "Fichero: %s (para escritura) -> ",
nombre );
   if( fichero )
      printf( "creado (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
```

```
    }

    fprintf( fichero, "Esto es un ejemplo de usar
la funcion \'fprintf\'\n" );
    fprintf( fichero, "\t 2\t 3\t 4\n" );
    fprintf( fichero, "x\tx\tx\tx\n\n" );
    for( i=1; i<=10; i++ )
       fprintf( fichero, "%d\t%d\t%d\t%d\n", i,
i*i, i*i*i, i*i*i*i );

    fprintf( stdout, "Datos guardados en el
fichero: %s\n", nombre );
    if( !fclose(fichero) )
       printf( "Fichero cerrado\n" );
    else
    {
       printf( "Error: fichero NO CERRADO\n" );
       return 1;
    }

    return 0;
}
```

Función fscanf ANSI C

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

Esta función recibe datos del stream apuntado por **stream**, bajo el control de la cadena apuntada por **formato** que especifica las secuencias de entrada permitadas y cómo han de ser convertidas para la asignación. Si hay argumentos insuficientes para el formato, el comportamiento no está definido. Si el formato termina mientras quedan argumentos, los argumentos restantes son evaluados (como siempre) pero ignorados. La función retorna control cuando el final de la cadena de formato es encontrado.

Valor de retorno:

La función *fscanf* retorna el número de datos de entrada asignados, que puede ser menor que ofrecido, incluso cero, en el caso de un error de asignación. Si un error de entrada ocurre antes de cualquier conversión, la función *fscanf* retorna el valor de la macro EOF.

Ejemplo:

```
#include <stdio.h>

int main()
{
   FILE *fichero;
   char nombre[10] = "datos.dat";
   unsigned int i, x1, x2, x3, x4;

   fichero = fopen( nombre, "r" );
   printf( "Fichero: %s (para lectura) -> ",
nombre );
   if( fichero )
      printf( "existe (ABIERTO)\n" );
   else
   {
      printf( "Error (NO ABIERTO)\n" );
      return 1;
   }

   printf( "Datos leidos del fichero: %s\n",
nombre );
   printf( "Esto es un ejemplo de usar la funcion
\'fprintf\'\n" );
   printf( "\t 2\t 3\t 4\n" );
   printf( "x\tx\tx\tx\n\n" );

   fscanf( fichero, "Esto es un ejemplo de usar la
funcion \'fprintf\'\n" );
   fscanf( fichero, "\t 2\t 3\t 4\n" );
   fscanf( fichero, "x\tx\tx\tx\n\n" );
   for( i=1; i<=10; i++ )
   {
      fscanf( fichero, "%d\t%d\t%d\t%d\n", &x1,
&x2, &x3, &x4 );
      printf( "%d\t%d\t%d\t%d\n", x1, x2, x3, x4
);
   }

   if( !fclose(fichero) )
      printf( "Fichero cerrado\n" );
   else
   {
```

```
        printf( "Error: fichero NO CERRADO\n" );
        return 1;
    }

    return 0;
}
```

Función fflush ANSI C

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.

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

Si **stream** apunta a un stream de salida o de actualización cuya operación más reciente no era de entrada, la función *fflush* envía cualquier dato aún sin escribir al entorno local o a ser escrito en el fichero; si no, entonces el comportamiento no está definido. Si **stream** es un puntero nulo, la función *fflush* realiza el despeje para todos los streams cuyo comportamiento está descrito anteriormente.

Valor de retorno:

La función *fflush* retorna cero si el stream fue despejado con éxito. Si se detectaron errores, entonces retorna EOF.

Ejemplo:

```
#include <stdio.h>

int main()
{
```

```
  char acumulador[BUFSIZ];

  setbuf( stdout, acumulador );

  printf( "Esto es una prueba\n" );
  printf( "Este mensaje se mostrara a la vez\n" );
  printf( "setbuf, acumula los datos en un
puntero\n" );
  printf( "hasta que se llene completamente\n" );

  fflush( stdout );

  return 0;
}
```

Función fseek ANSI C

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

La función *fseek* activa el indicador de posición de ficheros para el stream apuntado por **stream**. Para un stream binario, la nueva posición, medido en caracteres del principio del fichero, es obtenida mediante la suma de **desplazamiento** y la posición especificada por **origen**. La posición especificada es el comienzo del fichero si **origen** es SEEK_SET, el valor actual del indicador de posición de fichero si es SEEK_CUR, o final de fichero si es SEEK_END. Un stream binario realmente no necesita soportar llamadas a *fseek* con un valor de **origen** de SEEK_END. Para un stream de texto, o bien **desplazamiento** será cero, o bien **desplazamiento** será un valor retornado por una llamada anterior a la función ftell al mismo stream y **origen** será SEEK_SET. Una llamada correcta a la función *fseek* despeja el indicador de final de fichero para el stream y deshace cualquier efecto producido por la función ungetc en el mismo stream. Después de una llamada a *fseek*, la siguiente operación en un stream de actualización puede ser de entrada o salida.

Valor de retorno:

La función *fseek* retorna un valor distinto a cero sólo si una petición no se pudo satisfacer.

Ejemplo:

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

```
#include <string.h>

int main()
{
  char nombre[11] = "datos4.dat", mensaje[81]="";
  FILE *fichero;
  long int comienzo, final;

  fichero = fopen( nombre, "r" );
  printf( "Fichero: %s -> ", nombre );
  if( fichero )
    printf( "existe (ABIERTO)\n" );
  else
  {
    printf( "Error (NO ABIERTO)\n" );
    return 1;
  }

  if( (comienzo=ftell( fichero )) < 0 )   printf(
"ERROR: ftell no ha funcionado\n" );
  else    printf( "Posición del fichero: %d\n\n",
comienzo );

  fseek( fichero, 0L, SEEK_END );
  final = ftell( fichero );

  fseek( fichero, 0L, SEEK_SET );
  fgets( mensaje, 80, fichero );
  printf( "Tamaño del fichero \"%s\": %d bytes\n",
nombre, final-comienzo+1 );
  printf( "Mensaje del fichero:\n%s\n", mensaje );
  printf( "\nTamaño del mensaje (usando strlen):
%d\n", strlen(mensaje) );

  if( !fclose(fichero) )
    printf( "Fichero cerrado\n" );
  else
  {
    printf( "Error: fichero NO CERRADO\n" );
    return 1;
  }

  return 0;
```

```
}
```

Función ftell ANSI C

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

La función *fseek* obtiene el valor actual del indicador de posición de fichero para el stream apuntado por **stream**. Para un stream binario, el valor es el número de caracteres desde el principio del fichero. Para un stream de texto, su indicador de posición de fichero contiene información no especificado, servible a la función fseek para retornar el indicador de posición de fichero para el stream a su posición cuando se llamó a *ftell*; la diferencia entre los dos valores de retorno no es necesariamente una medida real del número de caracteres escritos o leídos.

Valor de retorno:

La función *ftell* retorna el valor del indicador de posición de fichero para el stream, si se tiene éxito. Si falla, la función *ftell* retorna **-1L** y guarda un valor positivo, según la definición de la implementación, en errno.

Ejemplo:

```
#include <stdio.h>
#include <string.h>

int main()
{
  char nombre[11] = "datos4.dat", mensaje[81]="";
  FILE *fichero;
  long int comienzo, final;

  fichero = fopen( nombre, "r" );
  printf( "Fichero: %s -> ", nombre );
  if( fichero )
    printf( "existe (ABIERTO)\n" );
  else
  {
    printf( "Error (NO ABIERTO)\n" );
    return 1;
  }
```

```
  if( (comienzo=ftell( fichero )) < 0 )   printf(
"ERROR: ftell no ha funcionado\n" );
  else    printf( "Posicion del fichero: %d\n\n",
comienzo );

  fseek( fichero, 0L, SEEK_END );
  final = ftell( fichero );

  fseek( fichero, 0L, SEEK_SET );
  fgets( mensaje, 80, fichero );
  printf( "Tama¤o del fichero \"%s\": %d bytes\n",
nombre, final-comienzo+1 );
  printf( "Mensaje del fichero:\n%s\n", mensaje );
  printf( "\nTama¤o del mensaje (usando strlen):
%d\n", strlen(mensaje) );

  if( !fclose(fichero) )
    printf( "Fichero cerrado\n" );
  else
  {
    printf( "Error: fichero NO CERRADO\n" );
    return 1;
  }

  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 librería 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:

```
2. struct stRegistro {
3.     char Valido;  // Campo que indica si el
  registro es válido
4.     char Nombre[34];
```

```
5.      int dato;
6.      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.

7. 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);
}
```