Lectura y escritura de un archivo binario

En esta guia vamos a hacer un programa en C que copie un archivo. Al hacerlo aprovecharemos para ver lo básico de lectura y escritura de archivos binarios en C.
En linux tenemos dos grupos de funciones para lectura y escritura de archivos. Las funciones **open()**, **write()**, **read()** y **close()** son de algo más bajo nivel y específicas de linux, así que no las usaremos en este ejemplo. En su lugar usaremos **fopen()**, **fwrite()**, **fread()** y **fclose()**, que son estándar de C y están presentes en todos los compiladores.

Abrir archivo

Lo primero de todo es abrir el archivo. Eso de abrir no es nada más que decirle al sistema operativo que prepare todo lo necesario para trabajar con un archivo concreto. Tendremos que abrir tanto el archivo que queremos leer como uno nuevo en el que escribiremos la copia del primero.
Un archivo tradicionalmente puede abrirse en **modo texto** o **modo binario**. No hay diferencia real entre uno y otro, salvo que en un archivo en modo texto se supone que hay fines de línea y en un momento dado hay funciones de archivos que pueden buscarlos. Si se abre en modo binario, la información puede ser de cualquier tipo y las funciones de archivos no buscarán fines de línea. Digo "tradicionalmente" porque en la práctica no hay ninguna diferencia entre abrirlo de una manera o de otra. Por convención, si el archivo es de texto legible se abre en modo texto. Si el archivo no es de texto legible, se abre en modo binario.
La función que nos permite abrir un archivo es **fopen()**. Sus parámetros son:

- Nombre del archivo que queremos abrir, con path absoluto o relativo.
- Modo de apertura (lo abrimos para leer, para escribir, etc). Si el modo pone b se abre en binario. Si no lo pone se abre en modo texto. Los posibles modos son:
    - **r** o **rb**: Abre el archivo para lectura. El archivo debe existir o tendremos un error.
    - **w** o **wb**. Abre el archivo para escribir en él. Puede o no existir. Si existe se guarda el contenido que hubiera. Si no existe se crea.
    - **a** o **ab**. Abre el archivo para añadirle datos al final. Respeta el contenido que tuviera.
    - **r+**, **rb+** o **r+b**. Abre el archivo para lectura y escritura (el + es el que indica que es para lectura y escritura a la vez). El archivo debe existir y respeta el contenido, aunque si escribimos algo iremos guardando datos.
    - **w+**, **wb+** o **w+b**. Abre el archivo para lectura y escritura. Crea el archivo si no existe y guarda el contenido si existe.
    - **a+**, **ab+** o **a+b**. Abre el archivo para lecturay y escritura. La escritura comenzará al final del archivo, respetando el contenido.

La función fopen() nos devuelve un FILE*. Esto no es más que un puntero a una estructura que contiene información sobre el archivo recién abierto. Normalmente podemos ignorar esa información, pero debemos hacer dos cosas con ese FILE* devuelto:

- Comprobar que no es NULL. Si es NULL es que ha habido algún problema y el archivo no se ha podido abrir (se ha abierto un archivo para leer que no existe, se ha abierto para escritura en un directorio en el que no tenemos permiso de escritura, etc).
- Guardarlo, porque es el puntero que requieren las demás funciones para saber sobre qué archivo escribir o leer. Nos servirá, desde el momento que lo abrimos, para identificar el archivo con el que queremos trabajar.

Para nuestro ejemplo, abriremos el archivo original como lectura y el archivo en el que haremos la copia como escritura. Puesto que nos da igual si el archivo es de texto o no, lo abriremos como binario, ya que es más general (un archivo de texto, con fines de línea, no es más que un caso particular de un archivo binario, que tiene bytes).

```
#include <stdio.h>


...


FILE *f1, *f2;
```

```
/* Apertura del archivo original, para lectura en binario*/
f1 = fopen ("archivo.dat", "rb");
if (f1==NULL)
{
  perror("No se puede abrir archivo.dat");
  return -1;
}

/* Apertura del archivo de destino, para escritura en binario*/
f2 = fopen ("archivo2.dat", "wb");
if (f2==NULL)
{
  perror("No se puede abrir archivo2.dat");
  return -1;
}
```

Ya está. f1 es el archivo origen y f2 es la copia.

Lectura del archivo binario

Para leer bytes de un archivo, sin importarnos si hay o no fines de línea, podemos usar la función **fread()**. Esta función tiene los siguientes parámetros

- **void *ptr**. Un puntero a una zona de memoria donde se guardarán los datos leidos del archivo. Nosotros pasaremos un array que hayamos creado previamente. En este array nos dejará la función los datos que leamos del archivo. El puntero es de tipo void para indicar que podemos pasar un puntero a cualquier tipo de dato que queramos. Nosotros pasaremos uno de char *.
- **size_t size**. El tamaño en bytes de los datos que vamos a leer. En nuestro caso, como queremos leer bytes, este valor será 1.
- **size_t nitems**. Numero de datos que queremos leer. El numero de datos que queremos leer multiplicado por el tamaño en bytes de cada dato, es decir, el número total de bytes a leer, no puede exceder el tamaño del array que pasemos como primer parámetro o tendremos efectos extraños en nuestro programa. Como el array que crearemos será de 1000 char, el numero de items maximo que podemos leer de golpe son 1000 (el tamaño del char es 1).
- **FILE *stream**. ¿Recuerdas lo que nos devolvía fopen()?. Pues tenemos que ponerlo aquí.

La función fread() devuelve el número de elementos leidos, que debería ser igual a nitems.

- Si en el archivo hay tantos o más elementos como los que hemos mandado leer con nitems, fread() los lee y devuelve nitems.
- Si no hay tantos elementos en el archivo, devolverá el número de elementos leidos que será menor que nitems.
- Si devuelve 0 es que no ha leido ningún elmento porque se ha llegado al final de archivo (se han terminado los datos del archivo).
- Si devuelve -1 es que ha habido algún error.

El código de lectura puede ser así

```
/* Para meter lo que vamos leyendo del archivo */
char buffer[1000];
/* Para guardar el número de items leidos o si ha habido error */
int leidos;
```

```
...


/* Lectura e if para detectar posibles errores */

leidos = fread (buffer, 1, 1000, f1);

if (leidos == ...)
```

Escribir un archivo binario

Para escribir un archivo binario, usamos la función **fwrite()**. Los parámetros son los mismos que fread(), pero con las siguientes salvedades

- **void *ptr**. El puntero a la zona de memoria. Debe estar relleno con los datos que queramos escribir en el archivo.
- **size_t size** y **size_t nitems** indican tamaño del elemento a escribir (1 en nuestro caso, ya que usamos char) y cuantos elementos queremos escribir. El número de elementos que queremos escribir son los que realmente hay en ptr, NO el tamaño del array ptr. Puede ser un array muy grande en el que sólo hemos rellenado tres elementos, así que nitems será tres.
- Consabido **FILE *stream**.

La función fwrite() nos devuelve el número de elementos escritos en el archivo o -1 si ha habido algún error.
En nuestro ejemplo, teniendo en cuenta que el número de elementos que tenemos disponibles en el array es el "leidos" que nos guardamos de la función fread(), el código de escritura quedaría así

```
fwrite (buffer, 1, leidos, f2);
```

Bucle hasta el fin de archivo

Ya sabemos lo básico para leer y escribir. Como queremos hacer una copia, debemos leer el archivo de entrada hasta que se termine e ir escribiendo en el archivo de salida. Para ello, debemos hacer un bucle hasta el final de archivo en el que se lean datos de un lado y se escriban en otro.
Aunque no la usaremos, hay una función **feof(FILE *stream)** a la que pasándole el dichoso FILE* del archivo, nos devuelve 0 si hay datos para leer u otro valor si el archivo se ha terminado. Por ello, un if bastaría para saber si hemos llegado al final de archivo

```
if (feof(f1))
  /* Hemos llegado al final de archivo */
```

O mejor todavía, se puede usar feof() como condición del bucle

```
while (!feof(f1))
{
  /* Hay datos disponibles para leer */
}
```

Hay un pequeño problema que suele ser metedura de pata común para los principiantes. Hasta que no hagamos una lectura, NO podemos saber si se ha terminado el archivo. La función fopen() sólo abre el archivo, no comprueba si hay datos o no. La función feof() no mira el archivo, sino simplemente si la últma lectura ha llegado o no al final del archivo. Si usamos feof() ANTES de hacer una lectura (con fread() en nuestro caso), SIEMPRE obtendremos que hay datos disponibles.
Por ello, es necesario hacer un fread() (o cualquier otra función de lectura) ANTES de empezar el bucle.

```
leidos = fread (buffer, 1, 1000, f1);
while (!feof(f1))
{
```

```
  /* Tratar los datos leídos en buffer */
  ...
  /* Y hacer otra lectura */
  leidos = fread (buffer, 1, 1000, f1);
}
```

Parece un poco feo, pero tampoco es elegante usar un do...loop, ya que deberíamos meter un if entre medias

```
do
{
  leidos = fread (buffer, 1, 1000, f1);

  /* Ahora hay que tratar los datos, pero si hemos llegado a final de archivo, NO hay datos que tratar. Hay que poner un fi para este caso */
  if (!feof(f1))
  {
    /* Tratar los datos leídos */
    ...
  }
} while (!feof(f1))
```

En fin, es cuestión de gustos y puedes usar la opción que más te guste o cualquier otra que se te ocurra. Lo importante es saber que hay que hacer al menos una lectura antes de utilizar feof().
Dije, de todas formas, que no iba a usar feof(). ¿Por qué?. Imagina que hay 10 bytes en el archivo y sólo 10 bytes. Cuando haga esta lectura

```
leidos = fread (buffer, 1, 1000, f1);
while (!feof (f1))
{
  /* tratar los datos leidos */
}
```

se leerán los 10 bytes y leidos tendrá 10. Pero como hemos llegado a final de archivo, feof() será cierto. Si me fio de feof() para terminar, no entraré en el bucle y no trataré esos 10 bytes.
En nuestro ejemplo, haremos el bucle mientras leidos sea mayor que 0. Si es cero, se ha acabado el archivo, si es -1 ha habido algún error.

```
leidos = fread (buffer, 1, 1000, f1);

/* Mientras se haya leido algo ... */
while (leidos!=0)
{
  /* ... meterlo en el archivo destino */
  fwrite (buffer, 1, leidos, f2);
  /* y leer siguiente bloque */
  leidos = fread (buffer, 1, 1000, f1);
```

```
}
```

Cerrar archivos

Cuando terminamos de usar un archivo, hay que cerrarlo con **fclose(FILE *)**. Unas tonterías sobre fclose() para que las tengas en cuenta.
Cuando se termina nuestro programa, el sistema operativo cierra todos los archivos que tengamos abiertos. En principio, para un programa sencillo como este, fclose() puede no ponerse sin problemas.
Sin embargo, conviene acostumbrarse a ponerlo para no meter la pata en programas más grandes y complejos. Hay dos posibles problemas si no se pone fclose().

- Hay un máximo de archivos que se pueden abrir a la vez. Si nuestro programa es grande, va abriendo archivos y los va dejando abiertos, llegará un momento en que alcanzaremos ese límite y tendremos un error. Este máximo es grande, pero no demasiado. Supongo que depende del sistema operativo concreto que usemos, pero puede ser del orden de 32, 64, etc. Vaya, que si nos despistamos, podemos llegar a él con cierta facilidad.
- Al cerrar el archivo nos aseguramos que su contenido realmente se esribe en el disco duro. Si nuestro programa no cierra el archivo, es posible que los datos no estén realmente en el disco duro y otros programas que vayan a acceder a ellos no los encuentren.

Así, que siguiendo las buenas constumbres...

```
fclose(f1);

fclose(f2);
```

Vamos ahora a un caso un poco más complejo, cómo acceder a cualquier posición del archivo de golpe.

Acceso aleatorio a un archivo

En el ejemplo de copia de un archivo binario vimos como abrir un archivo para lectura y escritura. También vimos como leer de él o escribir.
En ese ejemplo, tanto la lectura como la escritura eran secuenciales. Es decir, se abría el archivo desde el principio del archivo y se iban leyendo o escribiendo todos los datos del archivo en orden, desde el primero hasta el último, secuencialmente.
En la mayoría de los casos esto nos puede servir. Leemos el contenido completo del archivo y trabajamos con él. O abrimos el archivo para escribir los resultados y los vamos escribiendo todos seguidos.
Sin embargo, hay casos en que esto se nos puede quedar escaso. Hay veces en que necesitamos acceder una y otra vez al archivo para leer sus datos, modificar parte de ellos, por el medio. Imagina, por ejemplo, que tenemos guardado en un archivo nuestra agenda de teléfonos, con nuestros amigos. A veces podemos querer consultar uno que esté en medio del archivo, o cambiar su número de teléfono, o borrarlo como amigo porque nos hemos peleado.
C nos ofrece esta posibilidad. La función **fseek()** nos permite desplazarnos rápidamente, sin necesidad de ir leyendo todo, hasta una posición concreta de un archivo. Esto es lo que se conoce como **acceso aleatorio** a un archivo.

Definimos una estructura y rellenamos el archivo

Siguiendo con el ejemplo de la agenda, supón que tenemos una estructura de datos en C para guardar nuestros amigos. Como somos vaguetes, sólo pondremos dos campos, el nombre y el teléfono.

```
typedef struct {
   char nombre[20];
   int edad;
} persona;
```

Con esta estructura vamos a hacer un pequeño relleno de un archivo, para tener algo con lo que jugar. El código puede ser este

```
FILE *f1;
persona dato;
int i;

/* Abrimos el archivo binario y de escritura */
f1 = fopen ("persona.dat", "wb");
if (f1 == NULL)
{
   perror("No se puede abrir persona.dat");
   return -1;
}

/* Escribimos 10 datos, que serán
* Juan0, Juan1, Juan2, Juan3...
* con edad 0,1,2,3....
*/
for (i=0; i<10; i++)
```

```
{
   sprintf (dato.nombre, "Juan%d", i);
   dato.edad=i;
   /* El tamaño es sizeof(dato) y escribimos un registro */
   fwrite (&dato, sizeof(dato), 1, f1);
}


/* Cerramos el archivo */
fclose(f1);
```

A diferencia del código de ejemplo de copia de archivos, esta vez el tamaño del dato es sizeof(dato) mientras que en el ejemplo anterior era 1. En este caso escribimos un solo dato cada vez, mientras que antes escribiamos de 1000 en 1000.

Ir a un registro concreto para leer

Imaginemos ahora que una vez que tenemos el archivo, queremos leer la persona que ocupa la cuarta posición. Su índice es 3, ya que la numeración comienza por cero.
La función **fseek()** nos permite ir a una posición concreta. La función fseek() admite tres parámetros que son:

- **FILE *stream**. Este es el identificador para el archivo, que obtuvimos al hacer fopen().
- **long offset**. Esto es cuántos bytes queremos desplazarnos a partir de la posición que indica el siguiente parámetro.
- **int whence**. Este indica desde dónde queremos desplazarnos. Puede tener los siguientes valores:
    - **SEEK_SET**. Desde el principio del archivo
    - **SEEK_CUR**. Desde la posición actual en la que estemos.
    - **SEEK_END**. Desde el final del archivo.

Vamos a ver unos ejemplillos para que quede más claro.
Si ponemos fseek(f1, 100, SEEK_SET) nos situaremos en el byte 100 del archivo, empezando a contar desde el principio.
Si ponemos fseek(f1, 100, SEEK_END) nos situaremos 100 bytes antes del último byte del archivo.
Si acabamos de abrir el archivo y leemos cien bytes y luego hacemos fseek(f1, 100, SEEK_CUR), nos situaremos 100 bytes después del último byte leido, que era el 100, así que nos situaremos en el 200.
Volvemos al ejemplo de la agenda. Queremos el registro número 3 (que en realidad es el cuarto en el archivo). Lo primero que hay que hacer es calcular en que byte empieza ese registro. Esto es sencillo. Cada registro tiene sizeof(dato) bytes. El primero empezará en el 0, el segundo en sizeof(dato), el tercero en 2*sizeof(dato) y así sucesivamente.
Nuestro registro empieza en N*sizeof(dato) siendo N el número de registro, empezando a contar en cero. Para nuestro ejemplo, la posición es 3*sizeof(dato).
Sabiendo esto, está chupado irse a esa posición y leerla.

```
/* Abrimos el archivo para lectura y escritura. Más adelante querremos modificar el dato. */
f1 = fopen ("persona.dat", "rb+");
if (f1 == NULL)
{
   perror("No se puede abrir persona.dat");
   return -1;
}


/* Vamos al cuarto registro, indice 3 */
```

```
fseek (f1, 3*sizeof(dato), SEEK_SET);


/* Leemos y sacamos el resultado por pantalla */
fread (&dato, sizeof(dato), 1, f1);
printf ("nombre = %s\n", dato.nombre);
printf ("edad = %d\n", dato.edad);
```

Ya estamos en la posición justa, hemos leído y sacado por pantalla nuestro cuarto contacto.

Modificar los datos de un registro

Vamos ahora a modificar este mismo registro, porque lo de Juan3 no nos hace gracia y queremos que se llame Pedro. Además, su edad es 33 y no 3.
Puesto que acabamos de leer, estamos justo detrás del registro número 3. Así que debemos situarnos nuevamente al principio de este registro. Luego, simplemente rellenamos los datos a nuestro gusto y escribimos

```
/* Modificamos los datos en la estructura */
sprintf (dato.nombre, "Pedro");
dato.edad=33;


/* Nos volvemos a situar al principio del registro 3 */
fseek(f1, 3*sizeof(dato), SEEK_SET);


/* ¡¡ y lo guardamos !! */
fwrite (&dato, sizeof(dato), 1, f1);
```

Ya esta, se acaba de guardar nuestro amigo Juan3 y ahora se llama Pedro.

Saber cuántos amigos tenemos

Si queremos saber cuántos registros hay el archivo, la función **ftell()** nos puede ayudar. ftell() nos indica en qué posición en bytes del archivo estamos.
Si con fseek() nos vamos al final del archivo y luego con ftell() preguntamos dónde estamos, obtendremos el número de bytes del archivo, ya que estamos en el último. Dividiendo este número total de bytes entre lo que ocupa cada registro, sabremos cuántos registros tenemos.
El código puede ser así.

```
fseek(f1, 0, SEEK_END);
int numeroRegistros = ftell(f1)/sizeof(dato);
```

Borrar un registro

No insistas, no se puede. No podemos borrar un registro del archivo, ya que ocupa un espacio en bytes que seguirá ocupando.
¿Cómo borro entonces a alguien de mi agenda?
Tienes dos soluciones, la de conveniencia y la que tarda mucho.
La solución de conveniencia consiste en decidir que un determinado nombre o valor de edad indica que ese amigo no existe. Por ejemplo, un amigo sin nombre o de edad cero puede decir que ese registro se ha borrado. Cuando leamos la lista de amigos del archivo, antes de sacarlos por pantalla debemos comprobar si tiene o no edad cero y si la tiene, no lo sacamos por pantalla y vamos al siguiente. Si no hay ningún valor susceptible de convertirse en indicador de amigo no existente (por ejemplo, tenemos amigos sin nombre y de edad cero, así que no podemos usar eso), no nos

quedará más remedio que añadir un nuevo campo en nuestra estructura que indique si el amigo vale o no. Cuando queramos borrar un amigo, simplemente le ponemos un cero en la edad.
La solución que tarda mucho consiste en cuando queremos borrar un amigo, desplazamos todos los que van detrás una posición antes. Es decir, si queremos borrar al amigo 3, debemo coger al amigo 4 y ponerlo encima del 3, guardando. Luego el 5 encima del 4 y así sucesivamente. ¿Qué pasa con el último, el de la posición N?. Pues que queda duplicado. Esta en la posición N y en la N-1, ya que el de la N no podemos borrarlo. Debemos llevar, aparte, un entero que nos indique cuántos amigos tenemos. Si tenemos N amigos y borramos uno, tendremos N-1, por lo que el último, el N, que está duplicado "no vale".
Una buena solución suele ser una mezcla de ambas. Para el trabajo normal suponemos, por ejemplo, que edad cero quiere decir que ese amigo no existe. Cuando queremos añadir un amigo nuevo, buscamos uno de edad cero para guardarlo encima. Si no lo hay, ponemos el amigo al final. En un momento dado, bien a petición del usuario, bien porque de vez en cuando nuestro programa decide hacerlo, se "compacta" el archivo, haciendo todos los desplazamientos necesarios para eliminar amigos inexistentes.
Ahora que hemos hecho una cosa muy complicada, vamos con una sencilla. archivos de texto.

Archivos de texto en C

Ya hemos visto unos ejemplos de acceso aleatorio a un archivo binario. Vamos ahora con algo más simple, algunos ejemplillos de archivos de texto normalitos.
En primer lugar haremos una copia del archivo, usando las funciones **fgets()** y **fputs()**.
Después algún ejemplo simple y otro algo más complicado de uso de la función **fscanf()**, para ver que tiene más posibilidades de las que habitualmente aparecen en los libros de iniciación.

Funciones fgets() y fputs()

Vamos a hacer una copia de un archivo de texto. Aunque podemos copiarlo con el mismo programa que hicimos en el ejemplo de archivo binario, puesto que un archivo de texto no es más que un caso particular de archivo binario, vamos a hacerlo aquí de otra forma, específica para archivos de texto. Esto nos servirá de excusa para presentar las funciones **fgets()** y **fputs()**.
Lo primero, como siempre, será abrir los dos archivos. El original como archivo de lectura y la copia como archivo de escritura.

```
#include <stdio.h>

...

FILE *f = fopen("archivo.txt", "r");

FILE *f2 = fopen("archivo2.txt", "w");

if (f==NULL)

{

  perror ("Error al abrir archivo.txt");

  return -1;

}

if (f2==NULL)

{

  perror ("Error al abrir archivo2.txt");

  return -1;

}
```

Una de las posibles funciones que tenemos para leer un archivo de texto es **fgets()**. Esta función lee una línea completa del archivo de texto y nos la devuelve. Tiene los siguientes parámetros:

- **char *s**: Un array de caracteres en el que se meterá la línea leída del archivo. Este array debe tener tamaño suficiente para la línea o tendremos problemas.
- **int n**: Tamaño del array de caracteres. Si la línea es más larga, sólo se leeran n caracteres, para no desbordar el array. Si la línea tiene menos caracteres, se meteran en el array y se pondra un fin de cadena (un \0) al final.
- **FILE* stream**. Lo que obtuvimos con el fopen() del archivo que queremos leer.

Esta función devuelve la misma cadena de caracteres que le pasamos si lee algo del archivo o NULL si hay algún problema (fin de archivo, por ejemplo).
Para leer todas las líneas consecutivamente del archivo, podemos hacer un bucle hasta que esa función fgets() nos devuelva NULL. Puede ser así

```
char cadena[100]; /* Un array lo suficientemente grande como para guardar la línea más larga del archivo */

...

while (fgets(cadena, 100, f) != NULL)

{
```

```
  /* Aquí tratamos la línea leída */
}
```

Lo de distinto de NULL es una forma como otra cualquiera de mirar cuando se termina el archivo. Si queremos podemos hacer alguna variante usando feof() o cualquier otra cosa que se nos ocurra.
Un detalle que aquí no es importante, pero que es posible que tengamos que tener en cuenta en otros casos. El caracter fin de línea (\n en linux, \r\n en windows) NO se incluye en la cadena leída. fgets() lo lee del archivo, pero NO lo copia en cadena, así que no lo tenemos. En su lugar pone en en cadena un fin de cadena \0 (recuerda que en C todas las cadenas se terminan con un byte cero, que se representa como \0)
Como queremos ir escribiendo estas líneas en otro archivo, usaremos la función **fputs()**. Esta función escribe la cadena en el archivo y le añade un fin de línea (\n en linux, \r\n en windows).
La función fputs() admite los siguientes parámetros

- **char *s**: El array de caracteres que queremos escribir. Se escribirán secuencialmente todos los caracteres hasta que se encuentre un fin de cadena \0
- **FILE *stream**: Lo que obtuvimos al abrir el archivo de escritura.

Esta función devuelve el número de caracteres escritos o **FEOF** (un entero definido en algún include) si hay algún problema.
Nuestro código completo de copia quedaría así

```
char cadena[100]; /* Un array lo suficientemente grande como para guardar la línea más larga del archivo */
...
while (fgets(cadena, 100, f) != NULL)
{
  fputs(cadena, f2);
}
```

Ya sólo queda cerrar los archivos

```
fclose(f);
fclose(f2);
```

La función fscanf()

Una de las funciones que explica cualquier libro para leer archivos de texto es **fscanf()**.
La idea de fscanf es que como parámetro se le pasa el formato de lo que se quiere leer, y esta se encarga de buscarlo en el archivo y leerlo. Los parámetros de esta función son

- **FILE *stream**: Nuevamente, lo que obtuvimos con fopen() del ficheo en cuestión.
- **const char *format**: Formato de lo que hay en el archivo. Aquí, de una forma que vermos más abajo, se indica si en la línea hay un número, una cadena de caracteres, etc.
- **...** Varias variables o punteros a variables, donde fscanf() ira metiendo lo que vaya leyendo del archivo. El tipo de variables y cuántas variables pongamos aquí depende del formato que pongamos en el segundo parámetro.

Esta función devuelve el número de variables que ha conseguido rellenar.
Por ejemplo, si queremos leer una cadena de texto del archivo, ponemos

```
char cadena[100];
fscanf (f, "%s", cadena);
```

%s indica que estamos buscando una cadena de caracteres delimitada por espacios, tabuladores o saltos de línea y que queremos que la guarde en el array cadena. Por supuesto, la f es que la lea del archivo f, que es la misma f del fopen() que pusimos antes, al principio.
Si queremos un número entero, podemos poner

```
int valor;
fscanf (f, "%d", &valor);
```

%d indica que queremos un número entero. Hay que pasar la dirección de una variable entera (&valor en este caso) para que guarde ahí el entero que lea. En el caso de un array (como el de la cadena anterior), no hace falta poner el & delante.
De igual manera, tenemos los siguientes formatos:

```
int valor;



fscanf (f, "%i", &valor);
```

%i Un entero. Si empieza por 0x se lee en hexadecimal, si empieza por 0 se lee en octal y si empieza por otra cifra se lee como decimal. Por ejemplo, si en el archivo hay 0xFF, en valor se meterá un 255. Si en el archivo hay 010, en valor se meterá un 8 (en octal, el 8 se representa como 10).
%o Un octal. Se lee un número que se interpreta como octal. Si en el archivo hay 10, en valor se meterá 8.
%u Para un entero sin signo. valor debería ser de tipo unsigned int.
%x, **%X**: Un hexadecimal. Si en el archivo hay FF, en valor se meterá un 255
%f, **%e**, **%g**, **%E**: Para números con decimales. valor debería ser tipo float.
Hasta aquí lo que puedes encontrar en cualquier libro. Sin embargo, hay muchas más posibilidades y vamos a ver una de ellas.
En fscanf() podemos poner una expresión más compleja para tratar de coger el formato que queramos., indicando exactamente qué caracteres queremos leer. Imagina, por ejemplo, que nuestro archivo de texto es así

```
campo1:2:campo3

campo4:5:campo6
```

es decir, en cada línea tres campos separados por un separador, dos puntos en este caso. El primer campo es de texto, el segundo es un número y el tercero es de texto. Los campos de texto están formados por letras minúsculas y cifras.
Podemos leer una línea de esas de golpe, obteniendo los tres campos por separado, así

```
char cadena[100];

char cadena2[100];

int valor;

...

fscanf (f, "%[a-z0-9]:%d:%[a-z0-9]\n", cadena, &valor, cadena2);
```

Pero ... ¿ Qué es eso que hemos puesto ?
Básicamente son los tres campos, si te fijas, verás los separadores de : puestos ahí. Lo demás es una "representación" de qué cosas tiene cada campo (letras minúsculas y cifras el primero y tercero, un número el segundo).
El del medio, que es el más fácil, es un %d, que según vimos antes, corresponde a un entero. Es decir, estamos diciendo que el segundo campo es un número.
Para el primero y el tercero hemos puesto algo tan raro como esto %[a-z0-9]. Veamos que siginifica.
Podemos decir qué tipo de caracteres son los que componen un campo. Eso se pone con % y entre corchetes los caracteres válidos. Por ejemplo, si ponemos %[abc] quiere decir que el campo puede tener letras a, b y c. Si ponemos en el archivo abackk, entonces un fscanf() con el formato indicado leería sólo hasta "abac", dejándose "kk" sin leer.
Para no tener que escribir muchas letras, de la a a la z en nuestro caso, el formato admite que pongamos un guión. Así %[a-z] significa que valen todas las letras, de la a a la z, minúsculas.
Como nuestro campo también tiene números (campo**1**, etc), tenemos que poner que también valen los dígitos. Para ello, simplemente los ponemos y nuevamente evitamos escribirlos todos con un guión. Entonces %[a-z0-9] quiere decir que está compuesto por letras minúsculas y cifras.
El \n que pusimos al final del tercer campo es para indicar que detrás del tercer campo va un fin de línea.

Resumiendo, %[a-z0-9]:%d:%[a-z0-9]\n significa un campo cuyo valor es de letras minúsculas y cifras, un separador dos puntos, un campo numérico, otro separador dos puntos y un tercer campo compuesto por letras minúsculas y cifras, que además es el último de la línea.
Como puedes ver, con la función fscanf() podemos leer fácilmente archivos formateados de campos con separadores. De todas formas, este tipo de formatos es un poco delicadito y debemos estar muy seguros de que el archivo y el formato que pongamos coinciden exactamente. Si no es así, podemos leer valores muy raros y sin sentido.

Si echas un ojo al man de fsanf(), puedes ver más detalles sobre los posibles valores de este formato.

Archivos de texto en C++

Ya hemos visto cómo leer y escribir archivos de texto desde C. Eso vale perfectamente para C++, pero C++ tiene unas clases específicas para estas tareas: **ifstream**,**ofstream** y **fstream**. Vamos a ver aquí un ejemplo básico de cómo usarlas.

ifstream, ofstream y fstream

Las clases que nos da C++ para el acceso a archivos de texto son ifstream, ofstream y fstream. La primera es para leer del archivo (**i** de input, **f** de file y **stream**). La segunda es para escribir (**o** de output, **f** de file y **stream**). La tercera vale para las dos cosas, lectura y escritura.
Vamos a hacer un pequeño programa de copia de un archivo de texto. Lo primero, abrir los archivos, el archivo original para leer de él y un archivo nuevo para escribir en él.

Abrir los archivos

La apertura del archivo se puede hacer pasando parámetros al declarar la variable de estos tipos, o bien declarando la variable sin parámetros y luego llamando al método **open()**. Vamos a abrir cada uno de una manera distinta, para que veas ambas

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


int main()
{
  /* archivo original, se abre para lectura pasando parámetros en la declaración de la variable */
  ifstream f("archivo.txt", ifstream::in);


  /* archivo nuevo para copiar, se abre después de declararlo, llamando a open() */
  ofstream f2;
  f2.open("archivo2.txt", ofstream::out);
```

El primer parámetro es el nombre del archivo. Como siempre, path absoluto o relativo según nos interese. Para tus pruebas, sobre todo si usas algún IDE, te aconsejo path absoluto.
El segundo parámetros son unos flags para indicar cómo queremos abrir el archivo. Estos flags están en la clase **ios_base::openmode** y pueden ser los siguientes

- **app** (**app**end) Para añadir al final del archivo. Todas las escrituras que hagamos se iran al final del archivo. Si nos movemos por el archivo a una posición concreta (con el método seekp()) y luego escribimos, también se escribirá al final del archivo, ignorando la posición actual en la que nos situemos con seekp().
- **ate** (**at e**nd). Para añadir al final del archivo. Sin embargo, si nos movemos a otra posición del archivo (con seekp()) y escribimos, la escritura se hará en la posición en la que nos pongamos, y no al final como en el caso anterior.
- **binary** (**binary**) Se abre el archivo como archivo binario. Por defecto se abre como archivo de texto.
- **in** (**in**put) El archivo se abre para lectura.
- **out** (**out**put) El archivo se abre para escritura
- **trunc** (**trunc**ate) Si el archivo existe, se ignora su contenido y se empieza como si estuviera vacio. Posiblemente perdamos el contenido anterior si escribimos en él.

Si queremos abrir el archivo con varias de estas opciones, tenemos que componerlas con un OR (el caracter **|**), tal que así

```
f2.open("archivo2.txt", ofstream::out | ofstream::trunc);
```

Como estos flags son de una clase padre común, podemos acceder a ellos desde cualquier clase hija, por eso podemos poner ofstream::out o ifstream::in, e incluso ifstream::out aunque parezca que no tiene sentido.
Hay varias formas de comprobar si ha habido o no un error en la apertura del archivo. La más cómoda es usar el operator ! que tienen definidas estas clases. Sería de esta manera

```
if (!f)
{
  cout << "fallo" << endl;
  return -1;
}
```

Ese !f (no f) tan raro es el que nos dice si ha habido o no un error. !f devuelve true si ha habido algún problema de apertura del archivo.

Leer y escribir en el archivo

También tenemos muchos metodos específicos para leer y escribir bytes o texto: get(), getline(), read(), put(), write(), etc.
Sin embargo, los más curiosos son los operadores **<<** para escribir y **>>** para leer.

```
/* Declaramos un array con suficiente tamaño para leer las líneas */
char cadena[100];
...
/* Leemos */
f >> cadena;
...
/* y escribimos */
f2 << cadena;
```

Como pretendemos copiar un archivo, tendremos que meter esto en un bucle, leyendo una línea de un archivo y escribiéndola en otro, leyendo de uno y escribiendo en otro ... hasta que detectemos el final del archivo de entrada.
Para detectar el final del archivo de entrada tendemos el método **feof()**, que nos devuelve true si hemos inentado leer **después** del fin de archivo. He puesto después con negrita porque feof() NO devuelve true cuando estamos al final del archivo, sino cuando intentamos sobrepasarlo.
Por ello, no podemos hacer este bucle

```
/* Este bucle está mal */
while (!f.feof())
{
  f >> cadena;
  f2 << cadena;
}
```

Si el archivo está vacío, como no hemos hecho todavía ninguna lectura, feof() dirá que todavía no es el fin de archivo. Haremos una lectura que fallará (no hay nada que leer) y escribiremos "algo" en f2, lo que tuviera cadena antes de leer (puesto que no hemos leido nada).
Si el archivo no está vacío, todo funcionará aparentemente correcto, pero cuando lleguemos y hayamos escrito la última línea del archivo, feof() todavía dirá que no es el fin de archivo, así que será una nueva lectura, fallida. Escribiremos cadena otra vez con el contenido anterior. El resultado es que el archivo de salida contendrá dos veces la última línea.

Un posible bucle correcto es este

```
/* Hacemos una primera lectura */
f >> cadena;
while (!f.feof())
{
  /* Escribimos el resultado */
  f2 << cadena << endl;
  /* Leemos la siguiente línea */
  f >> cadena;
}
```

La primera lectura nos sirve para asegurarnos que feof() indica final de archivo y no entrar en el bucle si el archivo está vacío. Nos permite además cambiar el orden dentro del bucle, primero escribimos el dato que ya tenemos y luego intentamos leer la siguiente línea. Si no la hubiera, terminaría el bucle.
El **endl** que hemos puesto al final de la escritura es un fin de línea. Cuando leemos con f >> cadena, se lee hasta el final de línea, pero este no se incluye en cadena, sino que se descarta. Por ello, cuando escribmos, después de escribir cadena, necesitamos escribir el fin de línea.

Cerrar los archivos

Ahora queda lo más sencillo de todo, cerrar los archivos.

```
f.close();
f2.close();
```