introducción

archivos

El problema de los datos utilizados por un programa, es qué todos los datos se eliminan cuando el programa termina. En la mayoría de los casos se desean utilizar datos que no desaparezcan cuando el programa finaliza.

De cara a la programación de aplicaciones, un archivo no es más que una corriente (también llamada *stream*) de bits o bytes que posee un final (generalmente indicado por una **marca de fin de archivo**).

Para poder leer un archivo, se asocia a éste un **flujo** (también llamado secuencia) que es el elemento que permite leer los datos del archivo.

En C un archivo puede ser cualquier cosa, desde un archivo de disco a un terminal o una impresora. Se puede asociar un flujo a un archivo mediante una operación de apertura del archivo

jerarquía de los datos

La realidad física de los datos es que éstos son números binarios. Como es prácticamente imposible trabajar utilizando el código binario, los datos deben de ser reinterpretados como enteros, caracteres, cadenas, estructuras, etc.

Al leer un archivo los datos de éste pueden ser leídos como si fueran binarios, o utilizando otra estructura más apropiada para su lectura por parte del programador. A esas estructuras se les llama **registros** y equivalen a las estructuras (***structs***) del lenguaje C. Un archivo así entendido es una colección de registros que poseen la misma estructura interna.

Cada registro se compone de una serie de **campos** que pueden ser de tipos distintos (incluso un campo podría ser una estructura o un array). En cada campo los datos se pueden leer según el tipo de datos que almacenen (enteros, caracteres,...), pero en realidad son unos y ceros.

En la Ilustración 1 se intenta representar la realidad de los datos de un fichero. En el ejemplo, el fichero guarda datos de trabajadores. Desde el punto de vista humano hay salarios, nombres, departamentos, etc. Desde el punto de vista de la programación hay una estructura de datos compuesta por un campo de tipo String, un entero, un double y una subestructura que representa fechas.

El hecho de que se nos muestre la información de forma comprensible depende de cómo hagamos interpretar esa información, ya que desde el punto de vista de la máquina todos son unos y ceros.

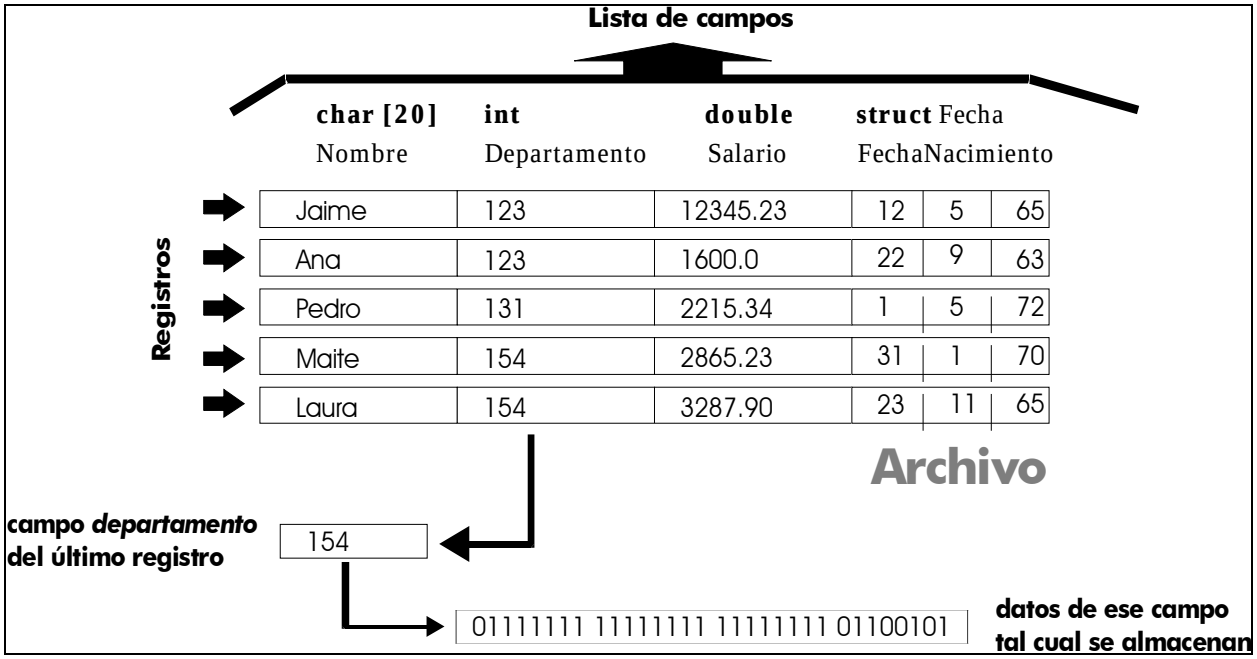


Ilustración 1, Ejemplo de la jerarquía de los datos de un archivo

clasificación de los archivos

por el tipo de contenido

- **Archivos de texto.** Contienen información en forma de caracteres. Normalmente se organizan los caracteres en forma de líneas al final de cada cual se coloca un carácter de fin de línea (normalmente la secuencia "\r\n"). Al leer hay que tener en cuenta la que la codificación de caracteres puede variar (la 'ñ' se puede codificar muy distinto según qué sistema utilicemos). Los códigos más usados son:
  - **ASCII**. Código de 7 bits que permite incluir 128 caracteres. En ellos no están los caracteres nacionales por ejemplo la 'ñ' del español) ni símbolos de uso frecuente (matemáticos, letras griegas,...). Por ello se uso el octavo bit para producir códigos de 8 bits, llamado ASCII extendido (lo malo es que los ASCII de 8 bits son diferentes en cada país).
  - **ISO 8859-1**. El más usado en occidente. Se la llama codificación de Europa Occidental. Son 8 bits con el código ASCII más los símbolos frecuentes del inglés, español, francés, italiano o alemán entre otras lenguas de Europa Occidental.
  - **Windows 1252**. Windows llama ANSI a esta codificación. En realidad se trata de un superconjunto de ISO 8859-1 que es utilizado en el almacenamiento de texto por parte de Windows.
  - **Unicode**. La norma de codificación que intenta unificar criterios para hacer compatible la lectura de caracteres en cualquier idioma. Hay varias posibilidades de aplicación de Unicode, pero la más utilizada en la actualidad es la UTF-8 que es totalmente compatible con ASCII y que usando el octavo bit con valor 1 permite leer más bytes para poder almacenar cualquier número de caracteres (en la actualidad hay 50000)

- **Archivos binarios.** Almacenan datos que no son interpretables como texto (números, imágenes, etc.).

por la forma de acceso

Según la forma en la que accedamos a los archivos disponemos de dos tipos de archivo:

- **Archivos secuenciales**. Se trata de archivos en los que el contenido se lee o escribe de forma continua. No se accede a un punto concreto del archivo, para leer cualquier información necesitamos leer todos los datos hasta llegar a dicha información. En general son los archivos de texto los que se suelen utilizar de forma secuencial.
- **Archivos de acceso directo.** Se puede acceder a cualquier dato del archivo conociendo su posición en el mismo. Dicha posición se suele indicar en bytes. En general los archivos binarios se utilizan mediante acceso directo.

Cualquier archivo en C puede ser accedido de forma secuencial o usando acceso directo.

estructura FILE y punteros a archivos

En el archivo de cabecera **stdio.h** se define una estructura llamada FILE. Esa estructura representa la cabecera de los archivos. La secuencia de acceso a un archivo debe poseer esta estructura.

Un programa requiere tener un puntero de tipo *FILE a cada archivo que se desee leer o escribir. A este puntero se le llama **puntero de archivos.**

apertura y cierre de archivos

apertura

La apertura de los archivos se realiza con la función **fopen**. Esta función devuelve un puntero de tipo FILE al archivo que se desea abrir. El prototipo de la función es:

```
FILE *fopen(const char *nombreArchivo, const char *modo)
```

nombreArchivo es una cadena que contiene la ruta hacia el archivo que se desea abrir. *modo* es otra cadena cuyo contenido puede ser:

modo	significado
"r"	Abre un archivo para lectura de archivo de textos (el archivo tiene que existir)
"w"	Crea un archivo de escritura de archivo de textos. Si el archivo ya existe se borra el contenido que posee.
"a"	Abre un archivo para adición de datos de archivo de textos
"rb"	Abre un archivo para lectura de archivos binarios (el archivo tiene que existir)
"wb"	Crea un archivo para escritura de archivos binarios (si ya existe, se descarta el contenido)
"ab"	Abre un archivo para añadir datos en archivos binarios

modo	significado
"r+"	Abre un archivo de texto para lectura/escritura en archivos de texto. El archivo tiene que existir
"w+"	Crea un archivo de texto para lectura/escritura en archivos de texto. Si el archivo tenía datos, estos se descartan en la apertura.
"a+"	Crea o abre un archivo de texto para lectura/escritura. Los datos se escriben al final.
"r+b"	Abre un archivo binario para lectura/escritura en archivos de texto
"w+b"	Crea un archivo binario para lectura/escritura en archivos de texto. Si el archivo tiene datos, éstos se pierden.
"a+b"	Crea o abre un archivo binario para lectura/escritura. La escritura se hace al final de el archivo.

Un archivo se puede abrir en **modo texto** o en **modo binario**. En modo texto se leen o escriben caracteres, en modo binario se leen y escriben cualquier otro tipo de datos.

La función **fopen** devuelve un puntero de tipo FILE al archivo que se está abriendo. En caso de que esta apertura falle, devuelve el valor NULL (puntero nulo). El fallo se puede producir porque el archivo no exista (sólo en los modos *r*), porque la ruta al archivo no sea correcta, porque no haya permisos suficientes para la apertura, porque haya un problema en el sistema,....

cierre de archivos

La función **fclose** es la encargada de cerrar un archivo previamente abierto. Su prototipo es:

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

pArchivo es el puntero que señala al archivo que se desea cerrar. Si devuelve el valor cero, significa que el cierre ha sido correcto, en otro caso se devuelve un número distinto de cero.

procesamiento de archivos de texto

leer y escribir caracteres

función getc

Esta función sirve para leer caracteres de un archivo de texto. Los caracteres se van leyendo secuencialmente hasta llegar al final. Su prototipo es:

```
int getc(FILE *pArchivo);
```

Esta función devuelve una constante numérica llamada EOF (definida también en el archivo **stdio.h**) cuando ya se ha alcanzado el final del archivo. En otro caso devuelve el siguiente carácter del archivo.

Ejemplo:

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

int main(){
    FILE *archivo;
    char c=0;
    archivo=fopen("c:\\prueba.txt","r+");
    if(archivo!=NULL) { /* Apertura correcta */
        while(c!=EOF){ /* Se lee hasta llegar al final */
            c=fgetc(archivo);
            putchar(c);
        }
        fclose(archivo);
    }
    else{
        printf("Error");
    }
    getch();
}
```

función fputc

Es la función que permite escribir caracteres en un archivo de texto. Prototipo:

```
int putc(int carácter, FILE *pArchivo);
```

Escribe el carácter indicado en el archivo asociado al puntero que se indique. Si esta función tiene éxito (es decir, realmente escribe el carácter) devuelve el carácter escrito, en otro caso devuelve EOF.

comprobar final de archivo

Anteriormente se ha visto como la función **fgetc** devuelve el valor EOF si se ha llegado al final del archivo. Otra forma de hacer dicha comprobación, es utilizar la función **feof** que devuelve verdadero si se ha llegado al final del archivo. Esta función es muy útil para ser utilizada en archivos binarios (donde la constante EOF no tiene el mismo significado) aunque se puede utilizar en cualquier tipo de archivo. Sintaxis:

```
int feof(FILE *pArchivo)
```

Así el código de lectura de un archivo para mostrar los caracteres de un texto, quedaría:

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

int main(){
    FILE *archivo;
    char c=0;
    archivo=fopen("c:\\prueba.txt","r+");
    if(archivo!=NULL) {
        while(!feof(archivo)){
            c=fgetc(archivo);
            putchar(c);
        }
    }
    else{
        printf("Error");
    }
    fclose(archivo);
    getch();
}
```

leer y escribir strings

función fgets

Se trata de una función que permite leer textos de un archivo de texto. Su prototipo es:

```
char *fgets(char *texto, int longitud, FILE *pArchivo)
```

Esta función lee una cadena de caracteres del archivo asociado al puntero de archivos *pArchivo* y la almacena en el puntero *texto*. Lee la cadena hasta que llegue un salto de línea, o hasta que se supere la longitud indicada. La función devuelve un puntero señalando al texto leído o un puntero nulo (NULL) si la operación provoca un error.

Ejemplo (lectura de un archivo de texto):

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

int main(){
    FILE *archivo;
    char texto[2000];
    archivo=fopen("c:\\prueba2.txt","r");
    if(archivo!=NULL) {
        fgets(texto,2000,archivo);
```

```
        while(!feof(archivo)){
           puts(texto);
           fgets(texto,2000,archivo);
        }
        fclose(archivo);
    }
    else{
        printf("Error en la apertura");
    }
}
```

En el listado el valor 2000 dentro de la funcón *fgets* tiene como único sentido, asegurar que se llega al final de línea cada vez que lee el texto (ya que ninguna línea del archivo tendrá más de 2000 caracteres).

función fputs

Es equivalente a la anterior, sólo que ahora sirve para escribir strings dentro del un archivo de texto. Prototipo:

```
int fputs(const char texto, FILE *pArchivo)
```

Escribe el texto en el archivo indicado. Además al final del texto colocará el carácter del salto de línea (al igual que hace la función **puts**). En el caso de que ocurra un error, devuelve EOF. Ejemplo (escritura en un archivo del texto introducido por el usuario en pantalla):

función fprintf

Se trata de la función equivalente a la función **printf** sólo que esta permite la escritura en archivos de texto. El formato es el mismo que el de la función **printf**, sólo que se añade un parámetro al principio que es el puntero al archivo en el que se desea escribir.

La ventaja de esta instrucción es que aporta una gran versatilidad a la hora de escribir en un archivo de texto.

Ejemplo. Imaginemos que deseamos almacenar en un archivo los datos de nuestros empleados, por ejemplo su número de empleado (un entero), su nombre (una cadena de texto) y su sueldo (un valor decimal). Entonces habrá que leer esos tres datos por separado, pero al escribir lo haremos en el mismo archivo separándolos por una marca de tabulación. El código sería:

```
#include <stdio.h>

int main(){
    int n=1; /*Número del empleado*/
    char nombre[80];
    double salario;
    FILE *pArchivo;
```

```
    pArchivo=fopen("c:\\prueba3.txt","w");
    if(pArchivo!=NULL){
        do{
            printf("Introduzca el número de empleado: ");
            scanf("%d",&n);
        /*Solo seguimos si n es positivo, en otro caso
        entenderemos que la lista ha terminado */
            if(n>0){
               printf("Introduzca el nombre del empleado: ");
               scanf("%s",nombre);
               printf("Introduzca el salario del empleado: ");
               scanf("%lf",&salario);
               fprintf(pArchivo,"%d\t%s\t%lf\n",
                                    n,nombre,salario);
            }
        }while(n>0);
        fclose(pArchivo);
    }
}
```

función fscanf

Se trata de la equivalente al **scanf** de lectura de datos por teclado. Funciona igual sólo que requiere un primer parámetro que sirve para asociar la lectura a un puntero de archivo. El resto de parámetros se manejan igual que en el caso de **scanf.**

Ejemplo (lectura de los datos almacenados con **fprintf**, los datos están separados por un tabulador).

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

int main(){
    int n=1;
    char nombre[80];
    double salario;
    FILE *pArchivo;
    pArchivo=fopen("c:\\prueba3.dat","r");
    if(pArchivo!=NULL){
        while(!feof(pArchivo)){
          fscanf(pArchivo,"%d\t%s\t%lf\n",&n,nombre,&salario);
          printf("%d\t%s\t%lf\n",n,nombre,salario);
        }
        fclose(pArchivo);
        getch();
    }
```

función fflush

La sintaxis de esta función es:

```
int fflush(FILE *pArchivo)
```

Esta función vacía el buffer sobre el archivo indicado. Si no se pasa ningún puntero se vacían los búferes de todos los archivos abiertos. Se puede pasar también la corriente estándar de entrada *stdin* para vaciar el búfer de teclado (necesario si se leen caracteres desde el teclado, de otro modo algunas lecturas fallarían).

Esta función devuelve 0 si todo ha ido bien y la constante EOF en caso de que ocurriera un problema al realizar la acción.

volver al principio de un archivo

La función **rewind** tiene este prototipo:

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

Esta función inicializa el indicador de posición, de modo que lo siguiente que se lea o escriba será lo que esté al principio del archivo. En el caso de la escritura hay que utilizarle con mucha cautela (sólo suele ser útil en archivos binarios).

operaciones para uso con archivos binarios

función *fwrite*

Se trata de la función que permite escribir en un archivo datos binarios del tamaño que sea. El prototipo es:

```
size_t fwrite(void *búfer, size_t bytes, size_t cuenta,FILE *p)
```

En ese prototipo aparecen estas palabras:

- **size_t**. Tipo de datos definido en el archivo **stdio.h,** normalmente equivalente al tipo **unsigned.** Definido para representar tamaños de datos.
- ***búfer*.** Puntero a la posición de memoria que contiene el dato que se desea escribir.
- ***bytes*.** Tamaño de los datos que se desean escribir (suele ser calculado con el operador **sizeof**)
- ***cuenta*.** Indica cuantos elementos se escribirán en el archivo. Cada elemento tendrá el tamaño en bytes indicado y su posición será contigua a partir de la posición señalada por el argumento *búfer*
- ***p*.** Puntero al archivo en el que se desean escribir los datos.

La instrucción **fwrite** devuelve el número de elementos escritos, que debería coincidir con el parámetro *cuenta*, de no ser así es que hubo un problema al escribir.

Ejemplo de escritura de archivo. En el ejemplo se escriben en el archivo *datos.dat* del disco duro C registros con una estructura (llamada *Persona*) que posee un texto para almacenar el nombre y un entero para almacenar la edad. Se escriben registros hasta un máximo de 25 o hasta que al leer por teclado se deje el nombre vacío:

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

typedef struct {
  char nombre[25];
  int edad;
}Persona;

int main(){
  Persona per[25];
  int i=0;
  FILE *pArchivo;

  pArchivo=fopen("C:\\datos.dat","wb");
  if(pArchivo!=NULL){
    do{
      fflush(stdin); /* Se vacía el búfer de teclado */
      printf("Introduzca el nombre de la persona: ");
      gets(per[i].nombre);
      if(strlen(per[i].nombre)>0){
        printf("Introduzca la edad");
        scanf("%d",&(per[i].edad));
        fwrite(&per[i],sizeof(Persona),1,pArchivo);
        i++;
      }
    }while(strlen(per[i].nombre)>0 && i<=24);
    fclose(pArchivo);
  }
  else{
    printf("Error en la apertura del archivo");
  }
}
```

La instrucción **fwrite** del ejemplo, escribe el siguiente elemento leído del cual recibe su dirección, tamaño (calculado con **sizeof**) se indica que sólo se escribe un elemento y el archivo en el que se guarda (el archivo asociado al puntero *pArchivo*).

función *fread*

Se trata de una función absolutamente equivalente a la anterior, sólo que en este caso la función lee del archivo. Prototipo:

```
size_t fread(void *búfer, size_t bytes, size_t cuenta,FILE *p)
```

El uso de la función es el mismo que **fwrite**, sólo que en este caso lee del archivo. La lectura es secuencial se lee hasta llegar al final del archivo. La instrucción **fread** devuelve el número de elementos leídos, que debería coincidir con el parámetro *cuenta*, de no ser así es que hubo un problema o es que se llegó al final del archivo (el final del archivo se puede controlar con la función **feof** descrita anteriormente).

Ejemplo (lectura del archivo **datos.dat** escrito anteriormente):

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

typedef struct {
  char nombre[25];
  int edad;
}Persona;

int main(){
  Persona aux;
  FILE *pArchivo;

  pArchivo=fopen("C:\\datos.dat","rb");
  if(pArchivo!=NULL){
    /* Se usa lectura adelantada, de otro modo
    el último dato sale repetido */
    fread(&aux,sizeof(Persona),1,pArchivo);
    while(!feof(pArchivo)){
      printf("Nombre: %s, Edad: %d\n",aux.nombre, aux.edad);
      fread(&aux,sizeof(Persona),1,pArchivo);
    }
    fclose(pArchivo);
  }
  else{
    printf("Error en la apertura del archivo");
  }
}
```

uso de archivos de acceso directo

función *fseek*

Los archivos de acceso directo son aquellos en los que se puede acceder a cualquier parte del archivo sin pasar por las anteriores. Hasta ahora todas las funciones de proceso de archivos vistas han trabajado con los mismos de manera secuencial.

En los archivos de acceso directo se entiende que hay un indicador de posición en los archivos que señala el dato que se desea leer o escribir. Las funciones **fread** o **fwrite** vistas anteriormente (o las señalas para leer textos) mueven el indicador de posición cada vez que se usan.

El acceso directo se consigue si se modifica ese indicador de posición hacia la posición deseada. Eso lo realiza la función **fseek** cuyo prototipo es:

```
int fseek(FILE * pArchivo, long bytes, int origen)
```

Esta función coloca el cursor en la posición marcada por el origen desplazándose desde allí el número de bytes indicado por el segundo parámetro (que puede ser negativo). Para el parámetro origen se pueden utilizar estas constantes (definidas en **stdio.h**):

- **SEEK_SET.** Indica el principio del archivo.
- **SEEK_CUR.** Posición actual.
- **SEEK_END.** Indica el final del archivo.

La función devuelve cero si no hubo problemas al recolocar el indicador de posición del archivo. En caso contrario devuelve un valor distinto de cero:

Por ejemplo:

```
typedef struct {
  char nombre[25];
  int edad;
}Persona;

int main(){
  Persona aux;
  FILE *pArchivo;

  pArchivo=fopen("C:\\datos.dat","rb");
  if(pArchivo!=NULL){
    fseek(pArchivo,3*sizeof(Persona),SEEK_SET);
    fread(&aux,sizeof(Persona),1,pArchivo);
    printf("%s, %d años",aux.nombre,aux.edad);
    fclose(pArchivo);
  }
}
```

El ejemplo anterior muestra por pantalla a la cuarta persona que se encuentre en el archivo (*3*sizeof(Persona)* devuelve la cuarta persona, ya que la primera está en la posición cero).

función *ftell*

Se trata de una función que obtiene el valor actual del indicador de posición del archivo (la posición en la que se comenzaría a leer con una instrucción del lectura). En un archivo binario es el número de byte en el que está situado el cursor desde el comienzo del archivo. Prototipo:

```
long ftell(FILE *pArchivo)
```

Por ejemplo para obtener el número de registros de un archivo habrá que dividir el tamaño del mismo entre el tamaño en bytes de cada registro. Ejemplo:

```
/* Estructura de los registros almacenados en
el archivo*/
typedef struct{
    int n;
    int nombre[80];
    double saldo;
}Movimiento;

int main(){
    FILE *f=fopen("movimientos3.bin","rb");
    int nReg;/*Guarda el número de registros*/

    if(f!=NULL){
        fseek(f,0,SEEK_END);
        nReg=ftell(f)/sizeof(Movimiento);
        printf("Nº de registros en el archivo = %d",nReg);
        getch();
    }
    else{
        printf("Error en la apertura del archivo");
    }
}
```

En el caso de que la función **ftell** falle, da como resultado el valor -1.

funciones *fgetpos* y *fsetpos*

Ambas funciones permiten utilizar marcadores para facilitar el trabajo en el archivo. Sus prototipos son:

```
int fgetpos(FILE *pArchivo, fpos_t *posicion);
int fsetpos(FILE *pArchivo, fpos_t *posicion);
```

La primera almacena en el puntero *posición* la posición actual del cursor del archivo (el indicador de posición), para ser utilizado más adelante pos **fsetpos** para obligar al programa a que se coloque en la posición marcada.

En el caso de que todo vaya bien ambas funciones devuelven cero. El tipo de datos **fpos_t** está declarado en la librería *stdio.h* y normalmente se corresponde a un número **long**. Es decir normalmente su declaración es:

```
typedef long fpos_t;
```

Aunque eso depende del compilador y sistema en el que trabajemos. Ejemplo de uso de estas funciones:

```
FILE *f=fopen("prueba.bin","rb+");
fpos_t posicion;
if(f!=NULL){
  ..... /*operaciones de lectura o de manipulación*/
  fgetpos(f,&posicion); /*Captura de la posición actual*/
  ... /*operaciones de lectura o manipulación */
  fsetpos(f,&posicion); /*El cursor vuelve a la posición
                          capturada */
```

ejemplos de archivos de uso frecuente

archivos de texto delimitado

Se utilizan muy habitualmente. En ellos se entiende que cada registro es una línea del archivo. Cada campo del registro se separa mediante un carácter especial, como por ejemplo un tabulador.

Ejemplo de contenido para un archivo delimitado por tabuladores:

```
1 Pedro      234.430000
2 Pedro      45678.234000
3 Pedro      123848.321000
5 Javier     34120.210000
6 Alvaro     324.100000
7 Alvaro     135.600000
8 Javier     123.000000
9 Pedro      -5.000000
```

La escritura de los datos se puede realizar con la función **fprintf**, por ejemplo:

```
fprintf(“%d\t%s\t\%lf”,numMov,nombre,saldo);
```

La lectura se realizaría con **fscanf:**

```
fscanf(“%d\t%s\t\%lf”,&numMov,nombre,&saldo);
```

Sin embargo si el delimitador no es el tabulador, podríamos tener problemas. Por ejemplo en este archivo:

```
AR,6,0.01
AR,7,0.01
AR,8,0.01
AR,9,0.01
AR,12,0.02
AR,15,0.03
AR,20,0.04
AR,21,0.05
BI,10,0.16
BI,20,0.34
BI,38,0.52
BI,57,0.77
```

La escritura de cada campo se haría con:

```
fprintf(“%s,%d,%lf”,tipo,modelo,precio);
```

Pero la instrucción:

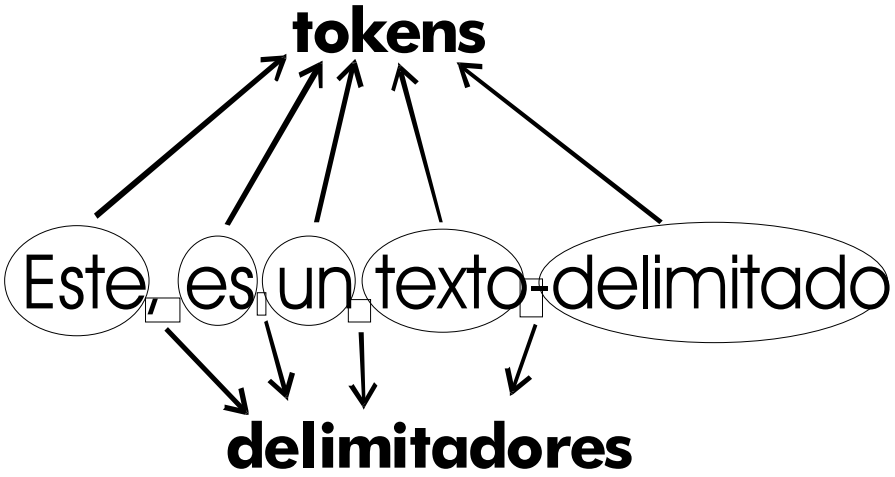
```
fscanf(“%s,%d,%lf”,tipo,&modelo,&precio);
```

da error, al leer *tipo* se lee toda la línea, no sólo hasta la coma.

La solución más recomendable para leer de texto delimitado es leer con la instrucción **fgets** y guardar la lectura en una sola línea. Después se debería utilizar la función **strtok**.

Esta función permite extraer el texto delimitado por caracteres especiales. A cada texto delimitado se le llama **token.** Para ello utiliza la cadena sobre la que se desean extraer las subcadenas y una cadena que contiene los caracteres delimitadores. En la primera llamada devuelve el primer texto delimitado.

El resto de llamadas a **strtok** se deben hacer usando NULL en el primer parámetro. En cada llamada se devuelve el siguiente texto delimitado. Cuando ya no hay más texto delimitado devuelve **NULL**.



En el diagrama anterior la frase "Este, es un texto-delimitado" se muestra la posible composición de *tokens* y delimitadores del texto. Los delimitadores se deciden a voluntad y son los caracteres que permiten separar cada *token*.

Ejemplo de uso:

```
int main(){
   char texto[] = "Texto de ejemplo. Utiliza, varios delimitadores\n\n";
   char delim[] = " ,.";
   char *token;

   printf("Texto inicial: %s\n", texto);

   /* En res se guarda el primer texto delimitado (token) */
   token = strtok( texto, delim);
   /* Obtención del resto de tokens (se debe usar NULL
   en el primer parámetro)*/
   do{
      printf("%s\n", token);
      token=strtok(NULL,delim);
   }while(token != NULL );
}
```

Cada palabra de la frase sale en una lína separada.

Para leer del archivo anterior (el delimitado con comas, el código sería):

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

/* Se encarga de transformar una línea del archivo de texto
en los datos correspondientes. Para ello extrae los tokens
y los convierte al tipo adecuado*/
void extraeDatos(char *linea, char *tipo, int *modelo,
             double *precio)  {
    char *cadModelo, *cadPrecio;
    strcpy(tipo,strtok(linea,","));
    cadModelo=strtok(NULL,",");
    *modelo=atoi(cadModelo);
    cadPrecio=strtok(NULL,",");
    *precio=atof(cadPrecio);
}

int main(){
    FILE *pArchivo=fopen("piezas.txt","r");
    char tipo[3];
    char linea[2000];
    int modelo;
    double precio;

    if(pArchivo!=NULL){
        fgets(linea,2000,pArchivo);
        while(!feof(pArchivo)){
            extraeDatos(linea,tipo,&modelo,&precio);
            printf("%s %d %lf\n",tipo,modelo,precio);
            fgets(linea,2000,pArchivo);
        }
        fclose(pArchivo);
    }
    getch();
}
```

archivos de texto con campos de anchura fija

Es otra forma de grabar registros de datos utilizando ficheros de datos. En este caso el tamaño en texto de cada fila es el mismo, pero cada campo ocupa un tamaño fijo de caracteres.

Ejemplo de archivo:

```
AR6  0.01
AR7  0.01
AR8  0.01
AR9  0.01
AR12 0.02
AR15 0.03
AR20 0.04
AR21 0.05
BI10 0.16
BI20 0.34
BI38 0.52
```

En el ejemplo (con los mismos datos vistos en el ejemplo de texto delimitado), los dos primeros caracteres indican el tipo de pieza, los tres siguientes (que son números) el modelo y los seis últimos el precio (en formato decimal):

En todos los archivos de texto de tamaño fijo hay que leer las líneas de texto con la función **fgets**. Y sobre ese texto hay que ir extrayendo los datos de cada campo (para lo cual necesitamos, como es lógico, saber la anchura que tiene cada campo en el archivo).

Ejemplo (lectura de un archivo de texto de anchura fija llamado **Piezas2.txt** la estructura del archivo es la comentada en el ejemplo de archivo anterior):

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>

/*Extrae de la cadena pasada como primer parámetro los c
caracteres que van de la posición inicio a la posición fin
(ambos incluidos) y almacena el resultado en el puntero
subcad*/
void subcadena(const char *cad, char *subcad,
                  int inicio, int fin){
    int i,j;
    for(i=inicio,j=0;i<=fin;i++,j++){
        subcad[j]=cad[i];
    }
    subcad[j]=0;/* Para finalizar el String */
}
```

```
/*Se encarga de extraer adecuadamente los campos de cada
línea del archivo */
void extraeDatos(char *linea, char *tipo,
                 int *modelo, double *precio)  {
    char cadModelo[3], cadPrecio[6];
    /* Obtención del tipo */
    subcadena(linea, tipo,0,1);
    /*Obtención del modelo */
    subcadena(linea, cadModelo,2,5);
    *modelo=atoi(cadModelo);
    /* Obtención del precio */
    subcadena(linea,cadPrecio,6,11);
    *precio=atof(cadPrecio);
}
int main(){
    FILE *pArchivo=fopen("piezas.txt","r");
    char tipo[3];
    char linea[2000];
    int modelo;
    double precio;

    if(pArchivo!=NULL){
        fgets(linea,2000,pArchivo);
        while(!feof(pArchivo)){
            extraeDatos(linea,tipo,&modelo,&precio);
            printf("%s %d %lf\n",tipo,modelo,precio);
            fgets(linea,2000,pArchivo);
        }
        fclose(pArchivo);
    }
    getch();
}/*fin del main */
```

formatos binarios con registros de tamaño desigual

Los ficheros binarios explicados hasta ahora son aquellos que poseen el mismo tamaño de registro. De tal manera que para leer el quinto registros habría que posicionar el cursor de registros con **fseek** indicando la posición como **4*sizeof(Registro)** donde *Registro* es el tipo de estructura del registro que se almacena en el archivo.

Hay casos en los que compensa otro tipo de organización en la que los registros no tienen el mismo tamaño.

Imaginemos que quisiéramos almacenar este tipo de registro:

```
typedef struct{
  char nombre[60];
  int edad;
  int curso;
} Alumno;
```

Un alumno puede tener como nombre un texto que no llegue a 60 caracteres, sin embargo al almacenarlo en el archivo siempre ocupará 60 caracteres. Eso provoca que el tamaño del archivo se dispare. En lugar de almacenarlo de esta forma, el formato podría ser así:

Tamaño del texto **(int)**	Nombre **(char [])**	Edad **(int)**	Curso **(int)**

El campo nombre ahora es de longitud variable, por eso hay un primer campo que nos dice qué tamaño tiene este campo en cada registro. Como ejemplo de datos:

7	Alberto	14	1
11	María Luisa	14	1
12	Juan Antonio	15	1

Los registros son de tamaño variable, como se observa con lo que el tamaño del archivo se optimiza. Pero lo malo es que la manipulación es más compleja y hace imposible el acceso directo a los registros (es imposible saber cuando comienza el quinto registro).

Para evitar el problema del acceso directo, a estos archivos se les acompaña de un **archivo de índices**, el cual contiene en cada fila dónde comienza cada registro.

Ejemplo:

Nº reg	Pos en bytes
1	0
2	20
3	46

En ese archivo cada registro se compone de dos campos (aunque el primer campo se podría quitar), el primero indica el número de registro y el segundo la posición en la que comienza el registro.

En los ficheros con índice lo malo es que se tiene que tener el índice permanentemente organizado.

indexados

Una de los puntos más complicados de la manipulación de archivos es la posibilidad de mantener ordenados los ficheros. Para ello se utiliza un fichero auxiliar conocido como índice.

Cada registro tiene que poseer una clave que es la que permite ordenar el fichero. En base de esa clave se genera el orden de los registros. Si no dispusiéramos de índice, cada

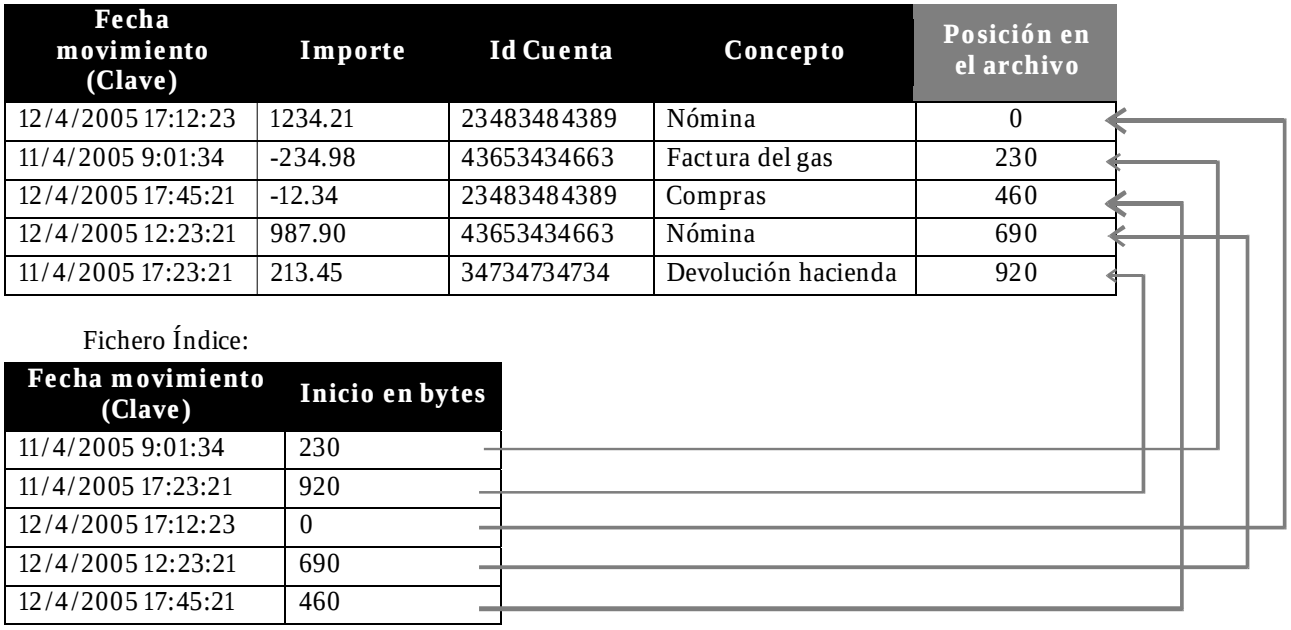
vez que se añade un registro más al archivo habría que regenerar el archivo entero (con el tiempo de proceso que consume esta operación).

Por ello se prepara un archivo separado donde aparece cada clave y la posición que ocupan en el archivo. Al añadir un registro se añade al final del archivo; el que sí habrá que reorganizar es el fichero de índices para que se actualicen, pero cuesta menos organizar dicho archivo ya que es más corto. Ejemplo de fichero de datos:

Fecha movimiento (Clave)	Importe	Id Cuenta	Concepto	Posición en el archivo
12/4/2005 17:12:23	1234.21	23483484389	Nómina	0
11/4/2005 9:01:34	-234.98	43653434663	Factura del gas	230
12/4/2005 17:45:21	-12.34	23483484389	Compras	460
12/4/2005 12:23:21	987.90	43653434663	Nómina	690
11/4/2005 17:23:21	213.45	34734734734	Devolución hacienda	920

Fichero Índice:

Fecha movimiento (Clave)	Inicio en bytes
11/4/2005 9:01:34	230
11/4/2005 17:23:21	920
12/4/2005 17:12:23	0
12/4/2005 12:23:21	690
12/4/2005 17:45:21	460



En el índice las claves aparecen ordenadas, hay un segundo campo que permite indicar en qué posición del archivo de datos se encuentra el registro con esa clave. El problema es la reorganización del archivo de índices, que se tendría que hacer cada vez que se añade un registro para que aparezca ordenado.

otros formatos binarios de archivo

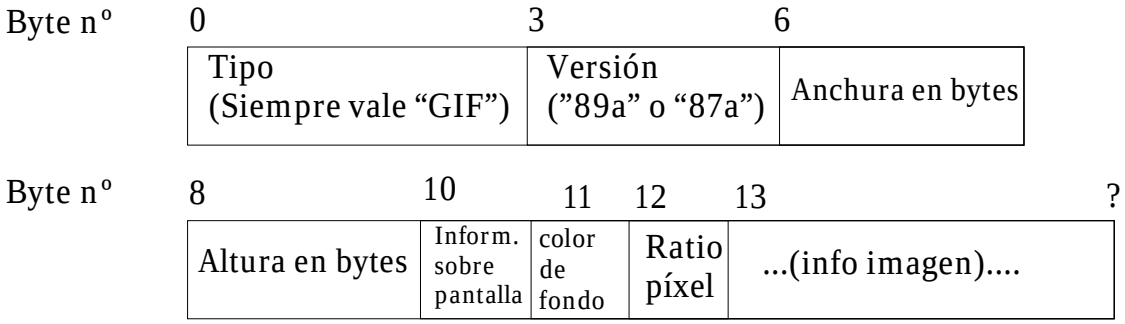
No siempre en un archivo binario se almacenan datos en forma de registros. En muchas ocasiones se almacena otra información. Es el caso de archivos que almacenan música, vídeo, animaciones o documentos de una aplicación.

Para que nosotros desde un programa en C podamos sacar información de esos archivos, necesitamos conocer el **formato de ese archivo.** Es decir como hay que interpretar la información (siempre binaria) de ese archivo. Eso sólo es posible si conocemos dicho formato o si se trata de un formato de archivo que hemos diseñado nosotros.

Por ejemplo en el caso de los archivos **GIF**, estos sirven para guardar imágenes. Por lo que lo que guardan son píxeles de colores. Pero necesitan guardar otra información al inicio del archivo conocida como cabecera.

Esta información tiene este formato:

Cabecera de los archivos GIF



Así para leer la anchura y la altura de un determinado archivo GIF desde un programa C y mostrar esa información por pantalla, habría que:

```
int main(){
    FILE *pArchivo;
    int ancho=0, alto=0; /* Aquí se almacena el resultado
    pArchivo=fopen("archivo.gif","rb");
    if(pArchivo!=NULL) {
    /* Nos colocamos en el sexto byte, porque ahí está la
    información sobre la anchura y después la altura*/
        fseek(pArchivo,6,SEEK_SET);
        fread(&ancho,2,1,pArchivo);
        fread(&alto,2,1,pArchivo);
        printf("Dimensiones: Horizontal %d, Vertical %d\n",
                                    ancho,alto);
    }
}
```

En definitiva para extraer información de un archivo binario, necesitamos conocer exactamente su formato.