



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Objetivo: Comprender el uso de archivos binarios y de texto para la persistencia de datos haciendo uso de las funciones proporcionadas por las bibliotecas de entrada/salida.

Introducción

Para iniciar este tema, daremos un vistazo al sistema de Entrada/Salida (E/S) de C definido por el estándar ANSI, que también se lo conoce como sistema de archivos con búfer. En él debemos saber que existen dos conceptos: flujo (**stream**) y archivo (**file**). El sistema de E/S provee al programador de una interfaz que es consistente e independiente del dispositivo con el cual se desea trabajar. Esto quiere decir que existe una capa de abstracción que permite tratar de la misma manera a los dispositivos de E/S. A esta abstracción se la denomina **Flujo**. Al dispositivo real se lo denomina **Archivo**. Vamos a ver cómo interactúan entre sí.

Flujos. El sistema de archivos de C permite trabajar con diferentes tipos de dispositivos, por ejemplo, discos, terminales, etc. Aunque cada uno de los dispositivos nombrados son diferentes, el sistema de archivos con búfer transforma cada uno de ellos en un dispositivo lógico llamado flujo. Todos los flujos se comportan de la misma forma. Esto se debe a que los flujos son independientes del dispositivo por lo cual la misma función puede escribir en un archivo de disco o en la consola (por ejemplo, la pantalla). Existen dos tipos de flujos: de texto y binarios (sólo bajo D.O.S. / Windows, a Unix / Linux le dan lo mismo).

Un **flujo de texto** es una secuencia de caracteres organizada en líneas terminadas por un carácter de **nueva línea**. En C, bajo el S.O. Windows, el carácter **nueva línea** es, en realidad, un par de caracteres: **retorno de carro (CR)** y **nueva línea (LF)** por lo que no hay una relación uno a uno entre los caracteres que se leen (escriben) y los del dispositivo externo. Le proponemos investigar qué sucede con la interpretación del carácter nueva línea en UNIX y Linux.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Un **flujo binario** es una secuencia de bytes que tienen una correspondencia uno a uno con un dispositivo externo.

Archivos. En el sistema de E/S basado en ANSI del lenguaje C, un archivo es un concepto lógico dentro del cual se puede encuadrar (casi) cualquier cosa, desde una terminal, una placa de red, o un archivo de disco. Tengamos en cuenta que C nació bajo el sistema operativo UNIX. Y en UNIX casi todos los dispositivos son tratados como archivos. Hay que tener en cuenta que no todos los archivos permiten las mismas operaciones, por ello se debe tener bien presente que todos los flujos son iguales, pero no todos los archivos lo son.

Puntero FILE. En un programa en C, cuando se opera sobre un archivo, se hace uso de un puntero de tipo **FILE** (se encuentra definida en la biblioteca **stdio.h**). El puntero apunta a una estructura que contiene la información necesaria para identificar y operar sobre el archivo en cuestión. Es importante destacar que cada archivo que se abra será identificado con un puntero de tipo **FILE**.

Apertura y cierre de archivos

Comencemos a ver los pasos necesarios para operar con archivos. Para ello, es necesario realizar la apertura del archivo. Además, se debe indicar el tipo y el modo de operación, dependiendo de tipo de archivo (binario o de texto) y la operación que se desee realizar sobre el archivo (lectura, escritura, agregado de registros). La tabla 1 muestra las combinaciones para realizar la apertura.

Apertura: Para abrir el archivo se utiliza la función **fopen**. Su prototipo es:

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

en dónde:

- **nombre:** puntero a char indicando el nombre del archivo que se desea abrir.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

- **modo**: puntero a char indicando el modo de apertura del archivo. La tabla 1 indica los modos de apertura.

La función supone la existencia del archivo (aunque esté vacío). En el caso de no existir, la función **fopen** falla y devuelve un valor nulo (**NULL**).

Cuando la apertura es exitosa se genera un espacio de almacenamiento en la que se almacena información del archivo abierto (de un compilador a otro para el mismo SO puede ser distinta y de un SO a otro, también), y cuando se lee, se graba o se posiciona con **fseek**, un indicador interno de la estructura del tipo **FILE** es el que lleva la posición dentro del archivo (cantidad de bytes desde el comienzo del archivo), el estado de dirección del flujo (E/S), estado de error, etc.

La tabla 1 indica los modos de apertura. Existen otros modos de apertura adicionales que utilizaremos cuando veamos acceso aleatorio y se encuentran resumidos en la tabla 2.

Modo	Significado	Respuesta de fopen
"rb", "rt"	Abre un archivo binario/texto para lectura.	Se ubica al inicio. Si no existe devuelve NULL.
"wb", "wt"	Crea un archivo binario/texto para escritura	Se ubica al inicio. Si ya existe, lo pisa y lo crea vacío.
"ab", "at"	Añade a un archivo binario/texto.	Se ubica al final. Si no existe, lo crea y se ubica, al inicio.

Tabla 1. Modos de apertura de archivos.

Modo	Significado	Respuesta de fopen
"r+b", "r+t"	Abre un archivo binario/texto para actualizar (leer/escribir)	Se ubica al inicio. Si no existe devuelve NULL.
"w+b", "w+t"	Crea un archivo binario/texto para actualizar (escribir/leer)	Se ubica al inicio. Si ya existe, lo pisa y lo crea vacío.
"a+b", "a+t"	Añade a un archivo binario/texto para actualizar (leer/escribir)	Se ubica al final. Si no existe, lo crea y se ubica, obviamente, al inicio.

Tabla 2 - Modos de apertura de archivos para acceso directo.

Cierre: Para cerrar el archivo se utiliza la función ***fclose***. Su prototipo es:

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

en dónde:

- ***stream***: puntero de tipo FILE al archivo que se desea cerrar.

Cualquier dato almacenado en el buffer del flujo que se encuentre pendiente de ser escrito en el archivo, será enviado al dispositivo para hacer efectiva su escritura. Luego libera la memoria ocupada por el ***fopen*** exitoso;

La función retorna cero si el stream fue cerrado de manera exitosa. En caso de que se detecten errores retorna EOF.

Ejemplo de apertura y cierre de un archivo

En el siguiente programa se ejemplifica el uso de la función ***fopen*** para crear un archivo binario y luego cerrarlo con la función ***fclose*** (Figura 1). El archivo se encontrará vacío, sin embargo, el hecho de que se encuentre vacío no impide que sea creado.

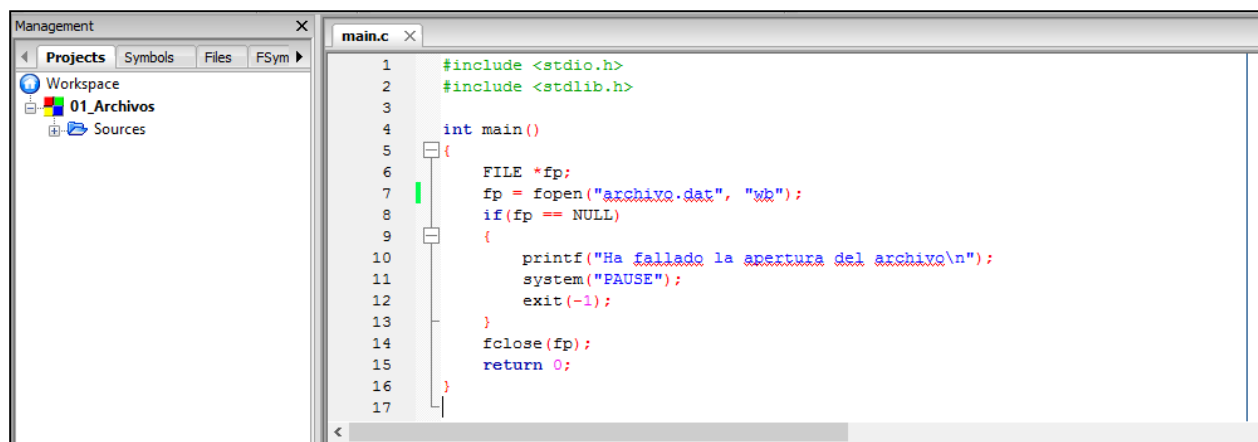


Figura 1. Ejemplo de apertura y cierre de un archivo binario.

Es momento de observar los asuntos relativos a la extensión del nombre del archivo y a la ruta en el disco en la cual el archivo generado será localizado.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

La extensión `.dat` del archivo no significa absolutamente nada. Bajo Windows, la extensión se utiliza para asociar un archivo a una aplicación y no como una forma de especificar el tipo de archivo. Por ello, se puede elegir la extensión que se desee (`dat`, `bin`, `aux`, `aaa`, `a01`, etc.), sin embargo, se sugiere que haga uso de una extensión que no confunda. Por ejemplo, no use `.txt` para un archivo binario (¡¡¡aunque técnicamente pueda hacerlo!!!).

La ruta destino del archivo dependerá de lo que se indique en el primer argumento de la función `fopen`, que como ya hemos dicho, corresponde al nombre del archivo. En el caso de indicar simplemente el nombre del archivo (como se hizo en el ejemplo de la Figura 1), el archivo se encontrará en la misma carpeta en la que se encuentra el archivo ejecutable del programa. Si desea especificar la ruta de destino, debe indicarse de manera explícita. Por ejemplo, si se desea abrir el archivo `alumnos.bin` que se encuentra en la carpeta `"C:\Programacion_1110\archivos"`, la forma de indicar la ruta sería: `"C:\\Programacion_1110\\archivos\\alumnos.bin"`. La doble barra invertida `"\\"` se utiliza para "escapar" la barra invertida `"\"`. Tenga en cuenta que las carpetas deben existir, de lo contrario la apertura fallará (`fopen` retornará `NULL` por no encontrar la carpeta destino).

Ejercicio propuesto: Modifique el programa de la Figura 1 para que cree el archivo binario en la carpeta `"C:\Programacion_1110\archivos"`. Pruebe el programa antes y después de crear las carpetas de destino. Haga uso de la herramienta de debug provista por Code::Blocks.

Lectura y escritura de archivos binarios

Ya hemos cubierto las operaciones para apertura y cierre de archivos. Ahora comenzaremos a ver las funciones que permiten la lectura y escritura sobre un archivo binario. Se debe recordar que la operación se realiza sobre el flujo vinculado al archivo



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

a través del puntero **FILE**. Ambas operaciones requieren que el archivo se encuentre abierto en el modo adecuado.

Supongamos que se ha abierto un archivo binario con una extensión de N bytes, en modo lectura o escritura (modos "rb" o "wb" de la Tabla 1). Si no han ocurrido errores, la función **fopen** habrá devuelto un puntero a **FILE** distinto de **NULL** (nuestro puntero para el ejemplo será **fp**). Esto ya lo hemos discutido antes, sin embargo, lo nuevo que deseamos destacar, desde el punto de vista conceptual, es que luego de la apertura del archivo, la estructura de control del tipo **FILE** se encuentra "apuntando al inicio" del archivo. La Figura 2 ejemplifica este caso.

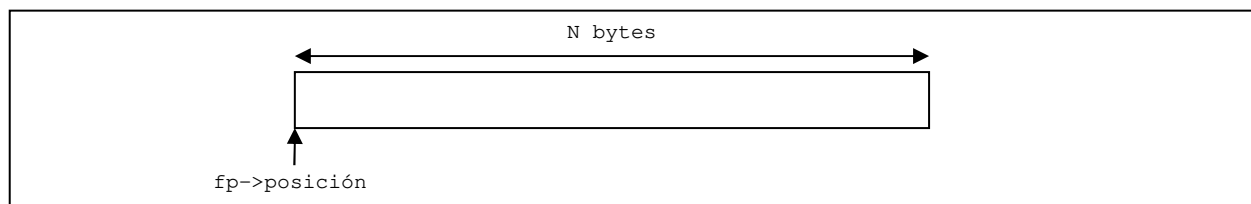


Figura 2. Posición del puntero **FILE *fp** luego de la apertura del archivo (modos *r* y *w*).

Veamos las funciones para lectura y escritura.

Lectura: Para leer desde un flujo conectado a un archivo se utiliza la función **fread**. Esta función lee del archivo la cantidad de bytes que se le indican en sus argumentos. Su prototipo es:

```
size_t fread(void *buffer_destino,  
             size_t num_bytes,  
             size_t cuenta,  
             FILE *fp);
```

en dónde:

buffer_destino: puntero a una zona de memoria en donde serán escritos los datos leídos del archivo.

num_bytes: cantidad de bytes a leer desde el archivo.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

cuenta: cantidad de elementos de tamaño **num_bytes** se van a leer.

fp: puntero al flujo conectado al archivo durante la operación de apertura del archivo.

Para determinar la cantidad de bytes a leer, la función utiliza los argumentos **num_bytes** y **cantidad**. La cantidad de bytes a leer será el resultado del producto **num_bytes * cantidad**.

La función retorna la cantidad de bytes leídos correctamente. En ese caso sería similar al producto anterior.

Es importante tener en cuenta las siguientes cuestiones. La primera, es que la función intentará leer del flujo la cantidad de bytes que se le especificó con los argumentos **num_bytes** y **cantidad** y los copiará a partir del inicio del área de memoria indicada por el puntero **buffer_destino**. La segunda, es que a la función no le interesa el tipo de dato que se intenta leer debido a que SÓLO interpreta que debe leer una determinada cantidad de bytes y los debe copiar en el destino que se le especifica. La tercera cuestión se encuentra centrada en la "posición" dentro del archivo indicada por la estructura de tipo **FILE** vinculada al flujo (***fp** del ejemplo). Cada vez que se realice una operación de lectura, el puntero quedará posicionado al inicio del siguiente registro. La Figura 3 muestra esa situación.

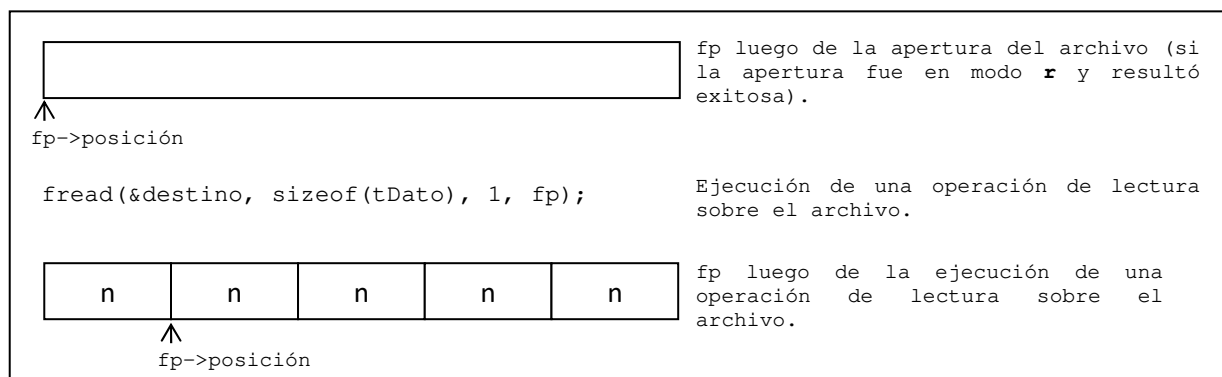


Figura 3. Cambio de la posición del puntero a FILE luego de una operación de lectura.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Suponemos que **n** es el tamaño en bytes de cada registro. En este caso la cantidad de bytes a leer se obtiene con el operador `sizeof(tDato)` y la cantidad es igual a 1. Como conclusión diremos que cada vez que se ejecute una operación de lectura, el indicador de posición dentro del archivo contenida en la estructura de tipo **FILE** utilizada para el manejo del archivo se desplazará hasta después de la cantidad de bytes leídos. Luego veremos qué sucede cuando se alcanza el final del archivo.

Escritura: Para escribir en un flujo conectado a un archivo se utiliza la función ***fwrite***. Esta función escribe en el archivo la cantidad de bytes que se le indican en sus argumentos. Su prototipo es:

```
size_t fwrite(void *buffer_origen,
              size_t num_bytes,
              size_t cuenta,
              FILE *fp);
```

en dónde:

buffer_origen: puntero a una zona de memoria desde donde serán obtenidos los datos a escribir en el archivo.

num_bytes: cantidad de bytes a escribir en el archivo.

cuenta: cantidad de elementos de tamaño **num_bytes** que se van a escribir.

fp: puntero al flujo conectado al archivo durante la operación de apertura del archivo.

Para determinar la cantidad de bytes a escribir, la función utiliza los argumentos **num_bytes** y **cantidad**. El número de bytes a escribir será el resultado del producto **num_bytes * cantidad**.

La función retorna la cantidad de bytes escritos correctamente.

Al igual que en el caso de la operación de lectura, se debe tener en cuenta que la función intentará escribir en el flujo la cantidad de bytes que se le especificó con los argumentos **num_bytes** y **cantidad** y que serán obtenidos a partir del área de memoria



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

indicada por el puntero `buffer_origen`. También debe tenerse en cuenta que a la función no le interesa el tipo de dato que se intenta escribir debido a que SOLO interpreta que deben escribir en el flujo una determinada cantidad de bytes. Por último, veamos qué sucede con el indicador de posición en la estructura del tipo **FILE** vinculada al flujo (`fp` del ejemplo). Cada vez que se realice una operación de escritura, el indicador de posición quedará ubicado al final del registro grabado. La Figura 3 muestra esa situación.

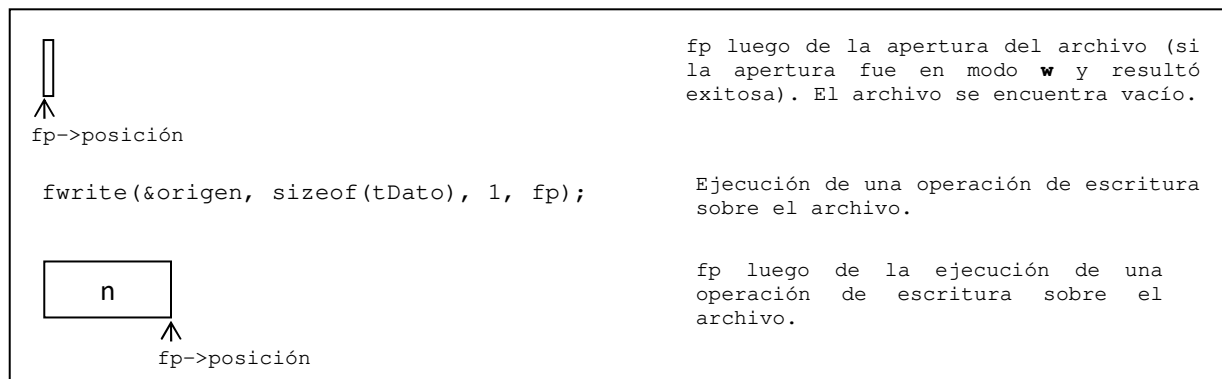


Figura 4. Cambio de la posición del puntero FILE luego de una operación de escritura.

Suponemos que `n` es el tamaño en bytes de cada registro. En este caso la cantidad de bytes a leer se obtiene con el operador `sizeof(tDato)` y la cantidad es igual a `1`. En este caso, como conclusión diremos que cada vez que se ejecute una operación de escritura, el puntero al archivo se desplazará hasta después de la cantidad de bytes leídos y el tamaño del archivo habrá crecido `n` bytes.

Ejemplo de escritura en un archivo binario

En el siguiente ejemplo se creará un archivo binario y se grabarán una serie de registros sobre dicho archivo, la Figura 5 contiene el código de la función que genera el archivo. En la Figura 6 se pueden ver la declaración de los tipos de datos utilizados en el ejemplo.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

1  #include "funciones.h"
2
3  int crearArchivo(char *nombreArchivo)
4  {
5      tEmpleado vectorEmpleados[] = {
6          {44444444, "Persona Cuatro", 'A', {1,4, 2004}, 44000.4F},
7          {22222222, "Persona Dos", 'B', {1,2, 2002}, 22000.2F},
8          {33333333, "Persona Tres", 'B', {1,3, 2003}, 33000.3F},
9          {55555555, "Persona Cinco", 'A', {1,5, 2005}, 55000.5F},
10         {11111111, "Persona Uno", 'C', {1,1, 2001}, 111000.1F}
11     };
12
13     FILE *fp = fopen(nombreArchivo, "wb");
14     if(fp)
15     {
16         fwrite(vectorEmpleados, sizeof(vectorEmpleados), 1, fp);
17         fclose(fp);
18         return 0;
19     }
20     return -2;
21 }

```

Figura 5. Creación de un archivo binario y escritura de datos sobre éste.

```

1  #ifndef TIPOS_DE_DATOS_H_INCLUDED
2  #define TIPOS_DE_DATOS_H_INCLUDED
3
4  typedef struct
5  {
6      int dia,
7      mes,
8      anio;
9  } tFecha;
10
11  typedef struct
12  {
13      long dni;
14      char apyn[36];
15      char categoria;
16      tFecha fecIngreso;
17      float sueldo;
18  } tEmpleado;
19
20
21  #endif // TIPOS_DE_DATOS_H_INCLUDED
22

```

Figura 6. Tipos de datos utilizados en el ejemplo.

Si se abre el archivo generado con un editor de textos (por ejemplo, el Block de Notas) podrá observar algo similar a lo que se muestra en la Figura 6, en donde solo se podrán ver de manera razonable los datos de tipo char (en el ejemplo serían los campos **apyn** y **categoria** del tipo de datos **tEmpleado**). Si desea explorar el contenido deberá utilizar un editor adecuado, por ejemplo, HxD.

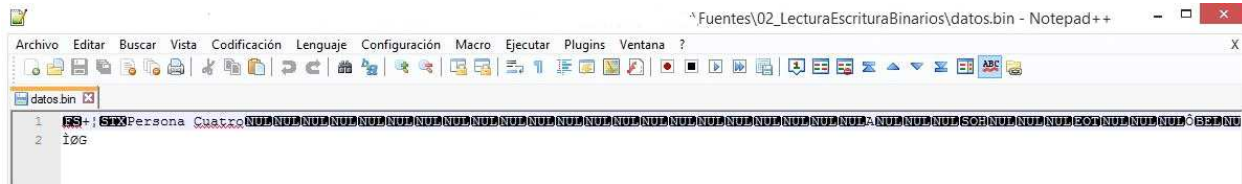


Figura 7. Vista del contenido del archivo con un editor de textos.

Ejemplo de lectura en un archivo binario

En el siguiente ejemplo se abrirá el archivo binario generado en el apartado anterior y se mostrará su contenido en la pantalla. Para ello, se lo abrirá en modo lectura y se lo recorrerá de manera secuencial. En la Figura 8 se puede ver la función **leerYmostrarArchivo** que se encarga de realizar la tarea requerida.

```

43  int leerYmostrarArchivo(char *nombreArchivo)
44  {
45      tEmpleado emp;
46      FILE *fp = fopen(nombreArchivo, "rb");
47
48      if(!fp)
49      {
50          return -1;
51      }
52
53      mostrarEmpleado(NULL);
54      fread(&emp, sizeof(tEmpleado), 1, fp);
55      while(!feof(fp))
56      {
57          mostrarEmpleado(&emp);
58          fread(&emp, sizeof(tEmpleado), 1, fp);
59      }
60      fclose(fp);
61      return 0;
62  }

```

Figura 8. Función para la lectura del archivo binario.

Es importante destacar lo que sucede entre las líneas 54 y 59. Se puede ver que en la línea 54 se hace una primera lectura. Luego se entra dentro del ciclo **while**. En la condición del **while** se invoca la función **feof** de la biblioteca **stdio.h**. Ésta función verifica que no se haya encontrado el final del archivo apuntado por el flujo, y esa es la condición de corte del ciclo. El prototipo de la función es el siguiente:

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



La función recibe el puntero a **FILE** que se va a verificar y retorna un valor distinto de 0 si no se ha alcanzado el final de archivo.

Volviendo a la lectura del archivo, si no se encontró el final del archivo se ingresa dentro del ciclo **while**. Una vez dentro, se invoca a la función **mostrarEmpleado** y se realiza una nueva lectura con **fread**. Se continúa así hasta que **feof** detecte el final del archivo apuntado por el flujo. La lectura de la línea 54 es necesaria antes de verificar que se haya encontrado el final del archivo y se la conoce como **lectura previa**. Si el archivo se encontrara vacío y se invocara a **feof** antes de esa lectura, **feof** no detectaría que se llegó al final, y es por ese motivo que se realiza la lectura previa. Luego de leer, el puntero a **FILE** estaría "apuntando" al final del archivo, y a partir de ese punto es cuando **feof** es capaz de detectar que se alcanzó el final del archivo apuntado por el flujo.

Por último, la Figura 9 muestra la función **mostrarEmpleado** que se encarga de la salida formateada al requerimiento.

```

23 void mostrarEmpleado(const tEmpleado *emp)
24 {
25     if(!emp)
26     {
27         printf("DNI      NOMBRE Y APELLIDO      CAT FEC. ING  SUELDO      \n");
28         printf("-----\n");
29         return;
30     }
31
32     printf("%08ld %-*.s %c  %02d/%02d/%04d %10.2f\n", emp->dni,
33           sizeof(emp->apyn)-1,
34           sizeof(emp->apyn)-1,
35           emp->apyn,
36           emp->categoria,
37           emp->fecIngreso.dia,
38           emp->fecIngreso.mes,
39           emp->fecIngreso.anio,
40           emp->sueldo);
41 }

```

Figura 9. Función **mostrarEmpleado** que permite mostrar en pantalla el tipo de dato **tEmpleado** formateado.

Sugerimos detenerse a ver en detalle cómo está diseñada la cadena de formato de la función **printf** utilizada en la función **mostrarEmpleado**.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Ejercicio propuesto: Tomando como base el archivo generado por la función `crearArchivo` (Figura 5), diseñe e implemente la función `buscarPorDNI` cuyo prototipo es el siguiente:

```
int buscarPorDNI(FILE *fp, tEmpleado *pe);
```

La función recibe un puntero al flujo correspondiente al archivo (en este caso, al recibir el puntero a `FILE` implica que el archivo ya se encuentra abierto en el modo adecuado), y un puntero a `tEmpleado` (`pe`). En el campo `dni` de la estructura (`pe->dni`) se recibirá el DNI a buscar en el archivo, que, en el caso de encontrarse presente, se deberán completar los demás campos de la estructura con los datos leídos desde el archivo y retornar 1. Caso contrario se retornará 0 y los demás campos de la estructura se dejarán como estaban.

Diseñe dos casos: en el primero, el archivo de datos debe encontrarse desordenado. En el segundo, el archivo se encontrará ordenado por DNI. Tenga en cuenta las consideraciones necesarias para efectuar el recorrido del archivo en cada caso (una pista puede ser no seguir recorriendo el archivo ordenado si el DNI buscado es mayor que el último leído).

Lectura y escritura de archivos de texto

Comenzaremos a tratar la lectura y escritura de archivos de texto. La principal diferencia con los archivos binarios es que al abrirllos con un editor de texto se podrán leer correctamente por un operador humano. En este caso, cada línea del archivo finalizará con un salto de línea.

Respecto a la operación de estos archivos, nuevamente haremos referencia a los modos de apertura que tratamos al inicio de este apunte en la Tabla 1. En este caso el archivo debe abrirse en el modo "`rt`" o "`wt`" según sea lectura o escritura.

Veamos las funciones para lectura y escritura.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Lectura: Para leer desde un flujo conectado a un archivo de texto utilizaremos la función `fgets`. La función lee caracteres y los guarda en `cad` hasta encontrar el carácter nueva línea (`'\n'`) o hasta leer `longitud - 1` caracteres, lo que ocurra primero, poniendo a continuación del último leído el carácter nulo (`'\0'`). El carácter `'\n'`, si se leyó, se guarda en la cadena `cad`. Su prototipo es:

```
char *fgets(char *cad,
            size_t longitud,
            FILE *fp);
```

en dónde:

`cad`: puntero a la memoria en donde será escrita la cadena leída desde el archivo.

`longitud`: cantidad máxima de caracteres a leer desde el archivo.

`fp`: puntero al flujo conectado al archivo durante la operación de apertura del archivo.

En caso de producirse un error o se llegue al final del archivo la función retornará **NULL**.

Escritura: Si bien la generación de un archivo de texto se puede realizar utilizando un editor de textos ASCII (Block de Notas, Notepad++, etc), la intención es que los podamos generar de manera programática a través de nuestras aplicaciones. Para ello utilizaremos la función `fprintf`. Esta función se comporta de manera similar a `printf` excepto que opera sobre un flujo conectado a un archivo. Su prototipo es:

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

en dónde:

`fp`: puntero al flujo conectado al archivo durante la operación de apertura del archivo.

`formato`: cadena de formato. El comportamiento es similar a los utilizados en `printf`.

La función retorna el número de caracteres realmente escritos, en caso de producirse un error retornará un número negativo.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Organización de los datos en archivos de texto

Ya hemos visto como leer y escribir en archivos de texto. Ahora es el turno de darle una mirada a la forma en que vamos a organizar los datos dentro del archivo. La idea es poder almacenar datos de interés de una manera lo suficientemente inteligente para luego poder explotar esos datos. Éstas no son ideas nuevas, ya que se vienen utilizando en la industria hace décadas.

En este curso veremos dos formas de organizar los datos en un archivo de texto: **campos de longitud variable** y **campos de longitud fija**. Al decir campos nos referimos a las diferentes unidades de datos que completan cada registro o fila del archivo.

En el caso de **campos de longitud variable**, cada campo se encuentra separado por un carácter en particular el cual se determina al momento del diseño. Poniendo como ejemplo el caso del tipo de dato tEmpleado, cada campo que lo conforma será separado por un determinado carácter, por ejemplo "|" entonces en el archivo los campos serán separados de la siguiente manera:

```
44444444|Persona Cuatro|A|1/4/2004|44000.44
22222222|Persona Dos|B|1/2/2002|22000.25
33333333|Persona Tres|B|1/3/2003|33000.32
55555555|Persona Cinco|A|1/5/2005|55000.50
11111111|Persona Uno|C|1/1/2001|111000.10
```

Se puede ver que todos los campos de los diferentes registros (cada fila), no se encuentran alineados. La ventaja de este método es que no hay desperdicio de espacio ya que la longitud del campo coincide con el dato.

Cuando estamos en el esquema de **campos de longitud fija**, la longitud de cada campo se encuentra definida con antelación y se respeta, aunque sobre espacio para almacenar el dato. En este caso, se adicionan caracteres para completar la longitud total del campo. Por ejemplo, si el campo **apyn** del tipo de dato **tEmpleado** es de 36 caracteres (contando uno más para el carácter nulo), y el nombre y apellido del



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

empleado ocupan sólo 20, se agregarán 15 espacios para completar la longitud total del campo en el archivo. Este esquema tiene como desventaja el desperdicio de espacio con los caracteres de relleno. Observen el ejemplo:

44444444	Persona Cuatro		A01042004	44000.44
22222222	Persona Dos		B01022002	22000.25
33333333	Persona Tres		B01032003	33000.32
55555555	Persona Cinco		A01052005	55000.50
01111111	Persona Uno		C01012001	111000.10

En el ejemplo se puede ver que el campo **apyn** se rellena con espacios en blanco, mientras que el campo **dni** se completa con ceros.

Ejemplo de escritura en un archivo de texto con campos de longitud variable

En el ejemplo que sigue se creará un archivo de texto y se grabará una serie de datos sobre éste. Los datos a grabar serán del tipo **tEmpleado**, y cada uno de los campos de la estructura estarán separados por el carácter ";". Se podría utilizar cualquier carácter, pero utilizaremos ";" porque el negocio no hace uso de este carácter dentro de sus datos (imagine que uno de los campos de la estructura fuera el correo electrónico y se utilizara el carácter "@" para separar los campos, se estaría añadiendo un campo extra que no existe con el dominio de la cuenta de email). El origen de los datos a grabar será el archivo binario creado en el ejemplo de escritura de archivos binarios. Es decir, la función leerá cada registro del archivo binario y lo escribirá en el archivo de texto separando cada campo con el carácter ";". En la Figura 10 se puede ver el código de la función que genera el archivo.

Si se abre el archivo generado con un editor de texto se podrá observar las diferencias en las longitudes entre los campos de diferentes registros y el carácter de separación entre los campos del registro (Figura 11).



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

66 int crearArchivoTextoLV(char *nombreArchivoOrigen, char *nombreArchivoDestino)
67 {
68     tEmpleado emp;
69     FILE *fpBin,
70         *fpTxtLV;
71
72     fpBin = fopen(nombreArchivoOrigen, "rb");
73     if(!fpBin)
74     {
75         return -1;
76     }
77
78     fpTxtLV = fopen(nombreArchivoDestino, "wt");
79     if(!fpTxtLV)
80     {
81         fclose(fpBin);
82         return -2;
83     }
84
85     fread(&emp, sizeof(tEmpleado), 1, fpBin);
86     while(!feof(fpBin))
87     {
88         fprintf(fpTxtLV,
89             "%ld;%s;%c;%d/%d/%d;%.2f\n",
90             emp.dni,
91             emp.apyn,
92             emp.categoria,
93             emp.fecIngreso.dia,
94             emp.fecIngreso.mes,
95             emp.fecIngreso.anio,
96             emp.sueldo);
97
98         fread(&emp, sizeof(tEmpleado), 1, fpBin);
99     }
100     fclose(fpBin);
101     fclose(fpTxtLV);
102     return 0;
103 }

```

Figura 10. Función que genera el archivo de texto de longitud variable.

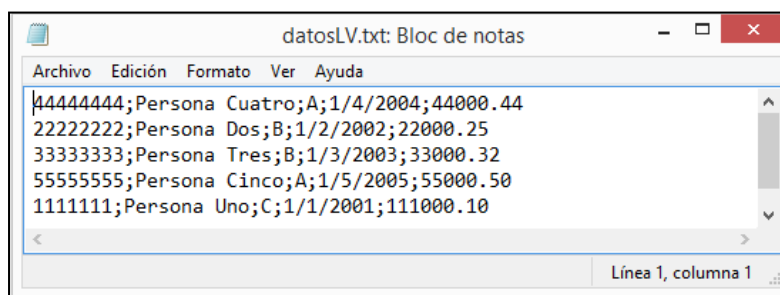


Figura 11. Contenido del archivo de texto de longitud variable.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Ejemplo de escritura en un archivo de texto con campos de longitud fija

La función indicada en la Figura 12 muestra el código para generar un archivo de campos de longitud fija. Preste especial atención a la máscara de formato de la función `fprintf`. La Figura 13 muestra el archivo de salida generado por la función.

```
105 int crearArchivoTextoLF(char *nombreArchivoOrigen, char *nombreArchivoDestino)
106 {
107     tEmpleado emp;
108     FILE *fpBin,
109         *fpTxtLF;
110
111     fpBin = fopen(nombreArchivoOrigen, "rb");
112     if(!fpBin)
113     {
114         return -1;
115     }
116     fpTxtLF = fopen(nombreArchivoDestino, "wt");
117     if(!fpTxtLF)
118     {
119         fclose(fpTxtLF);
120         return -2;
121     }
122     fread(&emp, sizeof(tEmpleado), 1, fpBin);
123     while(!feof(fpBin))
124     {
125         fprintf(fpTxtLF,
126             "%08ld%-*.s%c%02d%02d%04d%9.2f\n",
127             emp.dni,
128             sizeof(emp.apyn)-1,
129             sizeof(emp.apyn)-1,
130             emp.apyn,
131             emp.categoria,
132             emp.fecIngreso.dia,
133             emp.fecIngreso.mes,
134             emp.fecIngreso.anio,
135             emp.sueldo);
136
137         fread(&emp, sizeof(tEmpleado), 1, fpBin);
138     }
139     fclose(fpBin);
140     fclose(fpTxtLF);
141     return 0;
142 }
```

Figura 12. Función que genera un archivo de texto de campos de longitud fija.

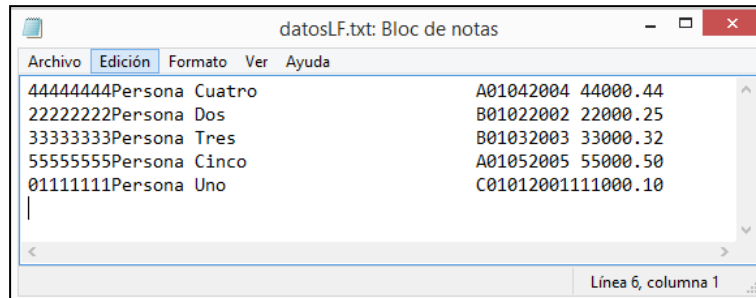


Figura 13. Contenido del archivo de texto de campos de longitud fija.

Ejercicio propuesto: En base a la definición de tipo de dato `tEmpleado` y al código que genera el archivo de texto de longitud fija determine la longitud en bytes de cada campo. ¿Cómo podría verificar su hipótesis?

Ejemplo de lectura de un archivo de texto

En la Figura 14 se puede observar la función que lee y muestra el contenido del archivo de texto de longitud variable generado anteriormente. Vea lo que sucede en el ciclo `while` (líneas 154 a 157). A diferencia de lo que se hacía en archivos binarios, cuando se hace el tratamiento de archivos de texto no se hace la lectura previa.

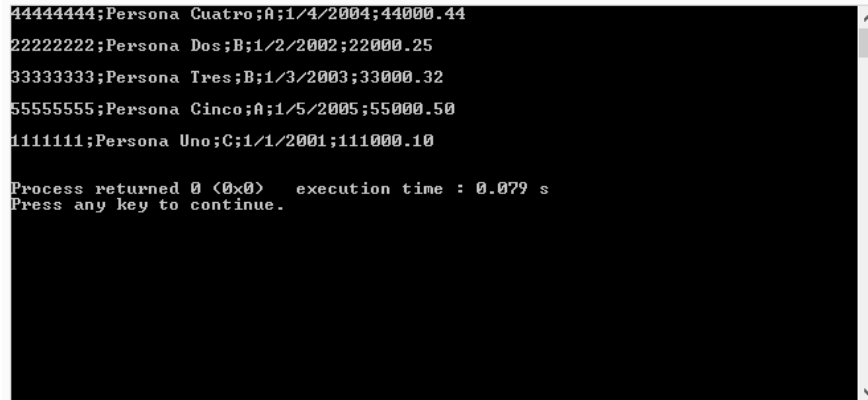
```

144  int leerYmostrarArchivoDeTexto(char *nombreArchivo)
145  {
146      char cad[100];
147      FILE *fp = fopen(nombreArchivo, "rt");
148
149      if(!fp)
150      {
151          return -1;
152      }
153
154      while(fgets(cad, sizeof(cad), fp))
155      {
156          printf("%s\n", cad);
157      }
158
159      fclose(fp);
160      return 0;
161  }
162

```

Figura 14. Función que realiza la lectura de un archivo de texto.

Esto sucede porque la función `fgets` retorna `NULL` cuando encuentra el final del archivo. La Figura 15 muestra la salida por pantalla.

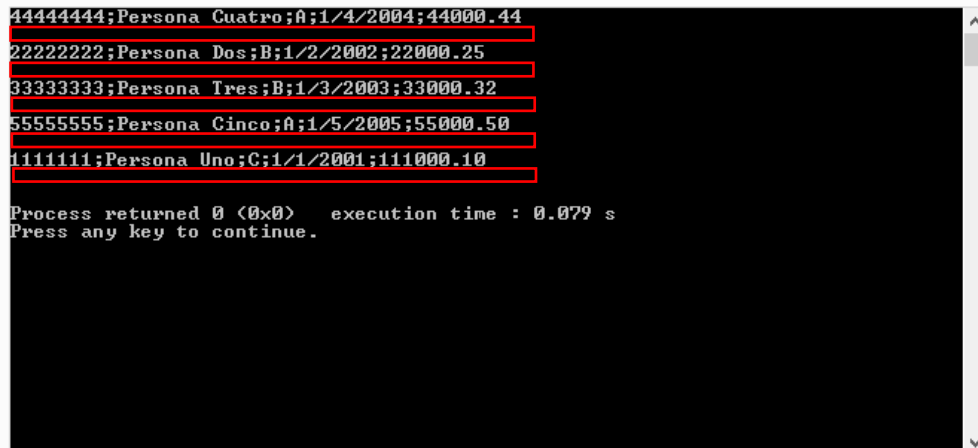


```
44444444;Persona Cuatro;A;1/4/2004;44000.44
22222222;Persona Dos;B;1/2/2002;22000.25
33333333;Persona Tres;B;1/3/2003;33000.32
55555555;Persona Cinco;A;1/5/2005;55000.50
11111111;Persona Uno;C;1/1/2001;111000.10

Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```

Figura 15. Salida por pantalla de la función que muestra un archivo de texto.

Ejercicio propuesto: Observe el código de la función indicada en la Figura 15 y responda: ¿Cuál es la causa que produce que cada línea de salida por pantalla tenga una nueva línea adicional (ver Figura 16)?



```
44444444;Persona Cuatro;A;1/4/2004;44000.44
22222222;Persona Dos;B;1/2/2002;22000.25
33333333;Persona Tres;B;1/3/2003;33000.32
55555555;Persona Cinco;A;1/5/2005;55000.50
11111111;Persona Uno;C;1/1/2001;111000.10

Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```

Figura 16. Determinar la causa (en rojo) por el cual aparece la nueva línea adicional.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Índice de figuras

<i>Figura 1. Ejemplo de apertura y cierre de un archivo binario.</i>	4
<i>Figura 2. Posición del puntero FILE *fp luego de la apertura del archivo (modos r y w).</i>	6
<i>Figura 3. Cambio de la posición del puntero a FILE luego de una operación de lectura.</i>	7
<i>Figura 4. Cambio de la posición del puntero FILE luego de una operación de escritura.</i>	9
<i>Figura 5. Creación de un archivo binario y escritura de datos sobre éste.</i>	10
<i>Figura 6. Tipos de datos utilizados en el ejemplo.</i>	10
<i>Figura 7. Vista del contenido del archivo con un editor de textos.</i>	11
<i>Figura 8. Función para la lectura del archivo binario.</i>	11
<i>Figura 9. Función mostrarEmpleado que permite mostrar en pantalla el tipo de dato tEmpleado formateado.</i>	12
<i>Figura 10. Función que genera el archivo de texto de longitud variable.</i>	17
<i>Figura 11. Contenido del archivo de texto de longitud variable.</i>	17
<i>Figura 12. Función que genera un archivo de texto de campos de longitud fija.</i>	18
<i>Figura 13. Contenido del archivo de texto de campos de longitud fija.</i>	19
<i>Figura 14. Función que realza la lectura de un archivo de texto.</i>	19
<i>Figura 15. Salida por pantalla de la función que muestra un archivo de texto.</i>	20
<i>Figura 16. Determinar la causa (en rojo) por el cual aparece la nueva línea adicional.</i>	20

Bibliografía

- [1] B. W. Kernighan y D. M. Ritchie, El lenguaje de programación C, Pearson Educación, 1991.
- [2] H. M. Deitel y P. J. Deitel, Cómo programar en C/C+, Pearson Educación, 1995.
- [3] Herbert Schildt, Turbo C/C++ 3.1 Manual de referencia, Osborne/McGraw-Hill, 1994.