

(Versión de fecha 16 de diciembre de 2024)

1 INTRODUCCIÓN

Hasta ahora, en el curso, todas las entradas y salidas de los programas las hemos hecho a través del terminal. No obstante, en programación es habitual que dichas entradas o salidas se hagan a través de ficheros¹ que están guardados en los dispositivos de almacenamiento del ordenador.

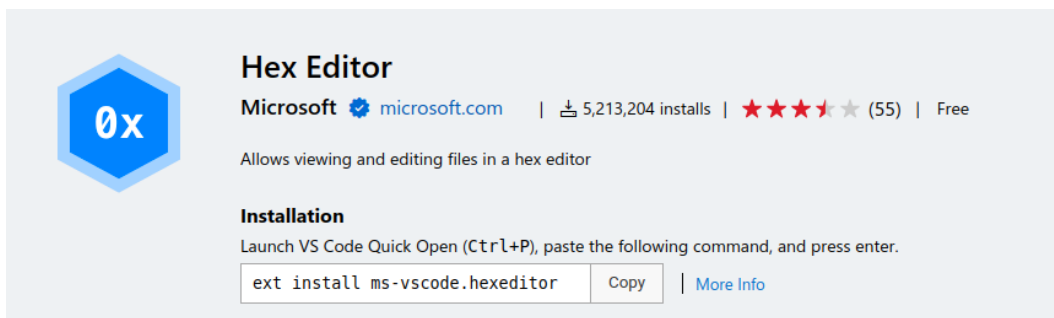
Como salida, los ficheros se utilizan para guardar de forma persistente la información generada por un programa. Esto permite poderlos utilizar con posterioridad por alguna aplicación para visualizarlos o hacer un procesamiento posterior.




Como entrada, los datos generados por algún programa pueden estar almacenados en disco en algún formato y nuestro programa podría procesarlos de alguna manera.

Se suelen considerar dos tipos de ficheros: ficheros de texto plano y ficheros binarios. Los primeros son, por ejemplo, los ficheros con el código de los programas que escribimos en C, pero que también se pueden utilizar para guardar otros tipos de información. Los ficheros binarios son, por ejemplo, los que guardan el código de los programas compilados aunque, al igual que sucede con los ficheros de texto, se pueden utilizar para guardar cualquier tipo de información.

En este curso solo vamos a explicar cómo guardar o recuperar información utilizando ficheros binarios.

Los ficheros de texto plano se pueden inspeccionar con cualquier editor de texto, por ejemplo, con el que proporciona VSCode. En cambio, los ficheros en formato binario no se pueden abrir con el editor de texto. Para inspeccionar su contenido hay que utilizar editores especiales. En VSCode puedes instalar la extensión *Hex Editor* (Editor hexadecimal), que permite abrir e inspeccionar al contenido de los ficheros en formato binario.



Hex Editor
Microsoft  microsoft.com |  5,213,204 installs |  (55) | Free

Allows viewing and editing files in a hex editor

Installation
Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter.

| [More Info](#)

2 OPERATIVA BÁSICA CON FICHEROS

Los programas no suelen hacer directamente operaciones de lectura o escritura en los dispositivos de almacenamiento. Lo que hacen es utilizar llamadas a servicios del sistema operativo, que es quien realmente

¹El término *fichero* y el término *archivo* se utilizan indistintamente en computación. En este texto hemos preferido utilizar la palabra *fichero*.

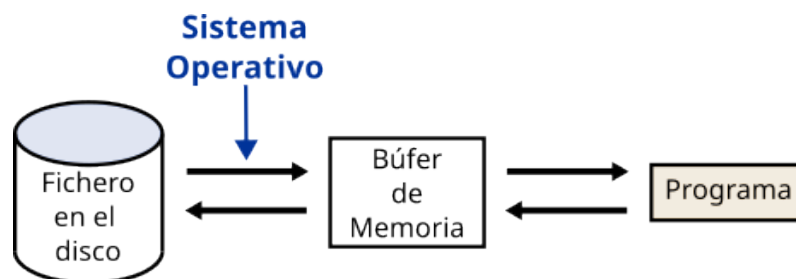
lee o escribe en los discos u otros dispositivos.

Estas operaciones de lectura o escritura son costosas en tiempo y recursos de procesamiento. Por ello, los programas suelen utilizar una zona de memoria en la que se van almacenando de manera provisional los datos que se leen o que se van a escribir en los ficheros. Esta zona de memoria se denomina el *búfer*, que será *de entrada* o *de salida*, según que se estén realizando operaciones de lectura o escritura.

Cuando un programa necesita leer datos desde un fichero que está en algún disco, el sistema operativo va llenando el *búfer de entrada* con los datos que hay en el disco y, a medida que el programa los consume, va actualizando el contenido del búfer con nuevos datos. Según el tamaño del búfer y el fichero, podría suceder que todos los datos del fichero se carguen en el búfer y, a partir de ese momento, no sean necesarios nuevos accesos al disco, pues el programa puede acceder a los datos directamente de la memoria.

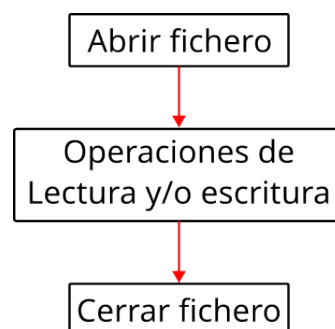
Cuando un programa tiene que escribir datos en disco, el mecanismo es similar, el programa va escribiendo datos en la memoria, en el búfer de salida, y el sistema operativo se encarga de volcar dichos datos periódicamente en el dispositivo de almacenamiento.

La figura siguiente esquematiza el mecanismo del búfer para operaciones de entrada o salida.



Un error que hay que prevenir en operaciones de escritura de ficheros es que el contenido del búfer no se escriba en el disco. Puede suceder que, si no se toman precauciones, parte de los datos que se querían escribir en un fichero se queden en el búfer y no lleguen al disco. Explicaremos cómo evitar este error.

El flujo de un programa cuando trabaja con ficheros es el que muestra la siguiente figura:



- **Abrir fichero:** esta operación es la que comunica al sistema operativo que se va a trabajar con determinado fichero del disco, para que habilite el correspondiente búfer de entrada o salida y realice los bloqueos correspondientes, de forma que otros programas no puedan acceder a dicho fichero mientras nuestro programa lo esté utilizando.
- **Operaciones de lectura o escritura:** en esta fase del programa se utilizarán instrucciones para leer datos del fichero o para escribir información en el mismo.

- **Cerrar fichero:** comunica al sistema operativo que hemos terminado de trabajar con el fichero. Si quedan datos en el búfer de salida, el sistema operativo los escribirá en el disco y dejará el fichero en condiciones de volver a ser utilizado por cualquier programa.

3 ABRIR FICHEROS

Como se ha dicho, se denomina *abrir un fichero* a comunicar al sistema operativo nuestra intención de utilizarlo desde nuestro programa. El sistema operativo necesita saber el nombre y la localización del fichero que queremos utilizar. Además, con el fin de poder habilitar el búfer correspondiente, el sistema operativo necesita saber si las operaciones que vamos a realizar son de lectura o escritura y si se trata de un fichero de texto o binario.

En el lenguaje C, la instrucción que se utiliza para abrir un fichero es *fopen()*, que tiene la siguiente signatura:

```
FILE* fopen(const char* nombre_fichero, const char* modo_apertura);
```

Los parámetros de *fopen()* son los siguientes:

- **nombre_fichero:** es una cadena de caracteres con el nombre completo del fichero. Si el fichero no está en el directorio de trabajo del programa, habrá que incluir la ruta completa al fichero.
- **modo_apertura:** es una cadena de texto que indica si el fichero se va a utilizar para lectura o escritura y si se trata de un fichero de texto o binario. En el Apartado 4 explicaremos cómo utilizar este parámetro.

La función *fopen()* devuelve un puntero a una estructura del tipo *FILE*. Esta estructura, que se suele denominar *manejador del fichero (handle)* o también *descriptor* del fichero, se define en la librería *stdio.h* y ofrece varios campos con datos acerca del fichero existente en el disco. Normalmente no utilizaremos ninguno de estos campos, lo que haremos es pasar el puntero a las instrucciones de lectura o escritura, para que puedan realizar su trabajo. Este puntero es la conexión de nuestro programa con el búfer habilitado por el sistema operativo.

Puede suceder que, al intentar abrir un fichero, la operación no tenga éxito. En ese caso, en lugar de un puntero al objeto *FILE*, el resultado de *fopen()* será *NULL*.

¡Importante!

Siempre que se hace *fopen()*, hay que comprobar que se ha obtenido un puntero válido al fichero con el que se quiere trabajar.

En el Apartado 5 se explica cómo hacer la comprobación correspondiente.

4 MODOS DE APERTURA DE LOS FICHEROS

Como se ha indicado en el Apartado 3, el segundo parámetro de la función *fopen()* es una cadena de texto indicando el *modo de apertura*. Este parámetro indica al sistema operativo el tipo de operaciones que vamos a hacer con el fichero y si se trata de un fichero de texto o binario.

La Tabla 4 muestra los valores más habituales de este parámetro.

Modos más habituales de apertura de ficheros

Modo	Descripción
"r"	Modo lectura (<i>read</i>). Abre un fichero de texto para lectura. El fichero tiene que existir, si no, la función <i>fopen()</i> devolverá <i>NULL</i> .
"w"	Modo escritura (<i>write</i>). Abre un fichero de texto para escribir en él. Si el fichero ya existe, se elimina y se escribe nuevo.
"a"	Modo añadir (<i>append</i>). Abre un fichero de texto para añadir contenido. Si el fichero existe, el nuevo contenido se añade al final del fichero. Si el fichero no existe, se crea y se escribe en él.
"rb"	Modo lectura para ficheros binarios.
"wb"	Modo escritura para ficheros binarios.
"ab"	Modo añadir para ficheros binarios.

Nota

En los modos para ficheros de texto que se muestran en la Tabla 4 se puede añadir la letra *t*, aunque no es necesario. Así, se tendría "*rt*", "*wt*" y "*at*".

Modos mixtos lectura-escritura

Los modos que se han indicado en la Tabla 4 solo permiten operaciones de lectura u operaciones de escritura. Si un fichero se abre para leer, no se podrán utilizar instrucciones de escritura en la misma sesión. Si un fichero se abre para escribir, no se podrán utilizar instrucciones de lectura.

Existen modos mixtos que permiten realizar operaciones de lectura y operaciones de escritura en la misma sesión. Son los modos "*r+*", "*w+*" y "*a+*" y sus equivalentes para ficheros binarios.

Aunque en determinadas ocasiones pueden resultar de utilidad, en general no se recomienda su uso, es mejor abrir un fichero para lectura o para escritura, no para hacer las dos cosas en la misma sesión.

5 COMPROBACIÓN DE QUE LA APERTURA HA TENIDO ÉXITO

Como se ha indicado, tras ejecutar la instrucción *fopen()* es necesario comprobar que el puntero que se ha recibido es válido. Si por algún motivo, la función *fopen()* no ha podido abrir el fichero, devolverá un valor *NULL* y las instrucciones posteriores de lectura o escritura en el fichero no tienen sentido.

Este error es más frecuente en los modos de lectura de ficheros, cuando el fichero del que se quiere leer no existe. Puede ser que el nombre del fichero no se haya indicado correctamente o que la ruta del fichero no es la que se ha escrito.

Cuando el nombre de un fichero del que se quiere leer se le solicita al usuario y el fichero no existe, se puede indicar el error y volver a solicitar al usuario un nombre correcto. En otras circunstancias, el puntero *NULL* nos obligará a terminar el programa.

Al programar, el puntero *FILE* se suele asignar a una variable. Son habituales nombres como *fid* (*file identifier*), *file*, *file_in*, *file_out* u otros parecidos que reflejan bien la intención de la variable.

El siguiente código muestra cómo podría ser el bloque de lectura de un fichero:

```
FILE fid = fopen("mi_fichero", "rb");

if(fid==NULL) {
    printf("Error al abrir fichero\n");
    return 1;
}

// Instrucciones si la apertura tuvo éxito
```

6 CIERRE DEL FICHERO

Normalmente, al finalizar un programa se cierran todos los ficheros que hubieran sido abiertos. Aún así, es bueno acostumbrarse a cerrar cualquier fichero que se hubiera abierto, tan pronto como se termina de trabajar con él. Ten en cuenta que, mientras se mantiene abierto un fichero, el riesgo de que pudiera resultar dañado es mayor. Por ejemplo, un corte de corriente podría invalidar el contenido del fichero y dejarlo inutilizable, pudiendo afectar incluso al propio dispositivo de almacenamiento.

Al cerrar un fichero, el sistema operativo realizará varias operaciones. Por ejemplo, al cerrar un fichero que se abrió para escritura, el sistema operativo escribirá en el disco cualquier dato que quedara aún en el búfer. También eliminará los bloqueos que hubiera podido imponer al fichero, dejándolo disponible para que otros programas lo puedan utilizar.

Expulsar USB

Para desconectar una memoria USB del ordenador, los sistemas operativos ofrecen una opción llamada *expulsar*, *desconectar con seguridad* u otro nombre parecido, según el sistema operativo del que se trate. En esa operación, el sistema operativo cerrará todos los ficheros que pudiera haber abiertos de forma que, aunque desconectemos el USB, no se produzcan daños en ningún fichero ni en el propio dispositivo.

Todos hemos desconectado alguna vez un USB sin realizar primero la expulsión segura. En muchas ocasiones no pasa nada, pero hay veces en las que el USB puede dañarse y se pueden perder los datos que tuviera almacenados.

A mí me ha pasado con memorias USB e incluso con un disco duro externo USB con cientos de Gigas de información irremplazable. El daño que produce una pérdida de datos importante es muy grande. Por ello, os aconsejo que siempre quitéis los USB mediante el procedimiento de expulsión segura que os proporcione el sistema operativo.

Para cerrar un fichero en C se utiliza la instrucción *fclose()*, que recibe como argumento un puntero al objeto *FILE* que se quiere cerrar.

El siguiente ejemplo muestra el esquema básico de utilización de un fichero, incluyendo la apertura, la comprobación de que el fichero se abrió correctamente y la operación de cierre:

```
// Abrir fichero
FILE fid = fopen("mi_fichero", "rb");

// Comprobación de que la apertura tuvo éxito
if(fid==NULL) {
    printf("Error al abrir fichero\n");
    return 1;
}

// Operaciones de lectura, en este caso, o de escritura
// si el fichero se hubiera abierto para escribir en él

// Cerrar fichero
fclose(fid);
```

En mi caso, cuando codifico un bucle o una bifurcación, escribo las llaves de apertura y cierre y luego completo dentro de ellas el contenido correspondiente. De la misma forma, cada vez que hago una instrucción *fopen()*, escribo la correspondiente instrucción *fclose()* y luego completo el contenido que haya que codificar entre ellas. Es una buena forma de no olvidar las llaves de cierre, en el caso de los bucles y bifurcaciones o la instrucción *fclose()*, cuando estoy trabajando con ficheros.

7 EL TIPO DE DATOS *SIZE_T*

Antes de hablar de las funciones *fread()* y *fwrite()* que utilizaremos para leer o escribir en un fichero, os voy a explicar el tipo de datos *size_t* y el operador *sizeof*, pues se utilizan profusamente en dichas funciones.

El lenguaje C no establece un estándar acerca del tamaño en bytes que debe tener el tipo *int*. En mi ordenador, el tipo *int* utiliza 4 bytes para almacenamiento, pero en otras plataformas puede utilizar otros tamaños².

Además, para cada tipo de números enteros, se puede usar la variante *unsigned*, que proporciona un mayor rango de valores, pero solo positivos o cero. Por ejemplo, si utilizas 1 byte para guardar un número entero, podrás guardar valores entre -128 y 127. En cambio, ese mismo byte, utilizado para enteros sin signo, permitiría guardar valores entre 0 y 255.

size_t significa *size type* y es un alias que está definido para referirse a un tipo de datos entero sin signo. Según la plataforma que se esté utilizando, el tipo *size_t* equivaldrá a un *unsigned int* o a un *unsigned long int*. En mi ordenador, el tipo *size_t* equivale a un *unsigned long int* y utiliza 8 bytes de almacenamiento.

Hay muchas funciones que utilizan el tipo *size_t* para sus parámetros o para el valor que devuelven. Por ejemplo, la función *strlen()* de la librería *string.h* devuelve un valor del tipo *size_t*.

Si se intenta escribir un valor *size_t* con la función *printf()* utilizando la especificación de formato *%i*, el compilador emitirá un aviso. Por ello, es mejor utilizar la especificación de formato *%lu* (*long unsigned*) o también *%zu* o *%ld*.

Otra forma de resolver este problema de formato es convertir el tipo *size_t* al tipo *int* o al tipo *unsigned int*, antes de imprimirlo³.

²La *plataforma*, en este contexto, se debe entender como el procesador, el sistema operativo y el compilador de C que se están utilizando

³En los casos habituales no suele existir problema con esta conversión, pero debemos estar seguros de que el número *size_t* que convertimos no sea mayor que el mayor número *int*.

Por ejemplo, si queremos imprimir el tamaño de una cadena de caracteres con el siguiente código, el compilador emitirá un *warning* de *especificación de formato incorrecta*:

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

int main() {
    char cad[] = "Hola mundo";
    printf("Tamaño: %i\n", sizeof(cad));
}
```

Podemos evitar el warning de dos formas:

- Ajustando la especificación de formato: usando `%lu`, `%ld` o `%ld`.
- Realizando una conversión implícita al tipo `int` antes de imprimir, como se muestra en el siguiente código.

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

int main() {
    char cad[] = "Hola mundo";

    int tam = strlen(cad);
    printf("Tamaño: %i\n", tam);
}
```

8 EL OPERADOR `sizeof`

El lenguaje C ofrece el operador `sizeof` que permite determinar el tamaño en bytes de una variable concreta o de un tipo de datos. El valor devuelto por el operador `sizeof` es del tipo `size_t`.

El siguiente ejemplo te puede servir para ver el funcionamiento del tipo `size_t` y del operador `sizeof` y, de paso, ver qué tamaño tienen en tu ordenador los tipos de datos enteros.

Ejemplo 1 Tamaño de algunos tipos de datos

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

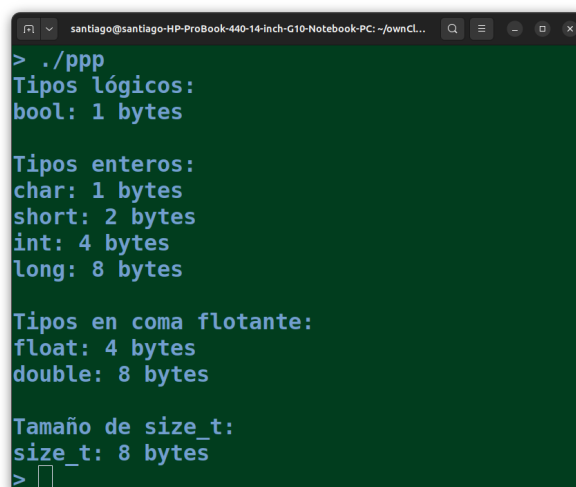
int main() {
    printf("Tipos lógicos:\n");
    printf("bool: %lu bytes\n", sizeof(bool));

    printf("\nTipos enteros:\n");
    printf("char: %lu bytes\n", sizeof(char));
    printf("short: %lu bytes\n", sizeof(short));
    printf("int: %lu bytes\n", sizeof(int));
    printf("long: %lu bytes\n", sizeof(long));

    printf("\nTipos en coma flotante:\n");
    printf("float: %lu bytes\n", sizeof(float));
    printf("double: %lu bytes\n", sizeof(double));

    printf("\nTamaño de size_t:\n");
    size_t n = 100;
    printf("size_t: %lu bytes\n", sizeof(n));

    return 0;
}
```



```
santiago@santiago-HP-ProBook-440-14-inch-G10-Notebook-PC: ~/ownCL...
> ./ppp
Tipos lógicos:
bool: 1 bytes

Tipos enteros:
char: 1 bytes
short: 2 bytes
int: 4 bytes
long: 8 bytes

Tipos en coma flotante:
float: 4 bytes
double: 8 bytes

Tamaño de size_t:
size_t: 8 bytes
> █
```

El operador *sizeof* se puede utilizar con otros tipos de datos, como las estructuras, los arrays o las cadenas de caracteres. En el caso de las cadenas de caracteres, es importante entender que el tamaño que devuelve *sizeof* no tiene por qué coincidir con el que devuelve la función *strlen()*. La función *strlen()* devuelve el número de caracteres de la cadena, mientras que *sizeof* devuelve el número de bytes que hay reservados para el array de caracteres.

El siguiente ejemplo te servirá para aclarar estos conceptos:

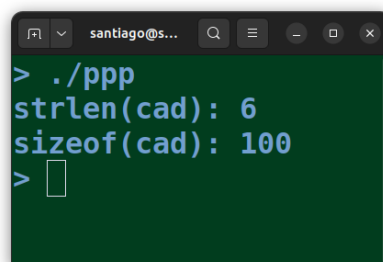
Ejemplo 2 Tamaño de cadenas de caracteres

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

int main() {

    char cad[100]= "Prueba";
    printf("strlen(cad): %lu\n", strlen(cad));
    printf("sizeof(cad): %lu\n", sizeof(cad));

    return 0;
}
```



```
> ./ppp
strlen(cad): 6
sizeof(cad): 100
> 
```

9 ESCRITURA DE FICHEROS BINARIOS

Para escribir información en formato binario se utiliza la instrucción *fwrite()*, cuya signatura es la siguiente:

```
size_t fwrite(const void* buffer, size_t tamano, size_t num_elementos, FILE* pFichero);
```

Los parámetros de *fwrite()* son los siguientes:

- **buffer:** es un puntero a la variable que vamos a escribir en el disco⁴.
- **tamano:** tamaño en bytes de cada uno de los elementos que vamos a escribir. El tipo de datos de este parámetro es *size_t*.
- **num_elementos:** numero de elementos que se van a escribir. El tipo de datos de este número también es *size_t*.
- **pFichero:** puntero al *FILE* donde se van a escribir los elementos.

La función *fwrite()* devuelve un *size_t* con el número de elementos que realmente se han escrito en el fichero, que estará entre cero y *num_elementos*.

Vamos a hacer un ejemplo, para que veas que es más sencillo de utilizar que de explicar por escrito:

⁴En realidad es un puntero a una zona de memoria de donde vamos a coger los bytes que escribiremos en el disco, pero en el curso utilizaremos un puntero a la variable que queremos escribir.

Ejemplo 3 Ejemplo de *fwrite()* con enteros

```
#include <stdio.h>

int main() {

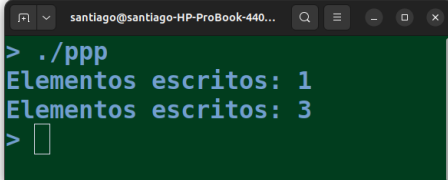
    // Abrir fichero y comprobar
    FILE* fid = fopen("Prueba_1.bin", "wb");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }

    // Escribir un número entero
    int x = 10;
    int result = fwrite(&x, sizeof(x), 1, fid);
    printf("Elementos escritos: %d\n", result);

    // Escribir un array de 3 enteros
    int nums[3] = {1, 2, 3};
    result = fwrite(nums, sizeof(int), 3, fid);
    printf("Elementos escritos: %d\n", result);

    // Cerrar fichero
    fclose(fid);

    return 0;
}
```



Observa el ejemplo anterior: primero se abre un fichero de nombre *Prueba_1.bin* en modo escritura y se comprueba que la apertura del fichero tuvo éxito; a continuación, se escribe el valor de la variable *x* con la siguiente instrucción:

```
int result = fwrite(&x, sizeof(x), 1, fid);
```

Observa que, el número de elementos escritos que devuelve la instrucción *fwrite()*, lo recogemos en una variable del tipo entero. Esto se puede hacer y facilita su utilización posterior.

Observa cómo se ha indicado el puntero a la variable que queremos escribir, cómo se indica su tamaño utilizando el operador *sizeof* y cómo se ha indicado el número de elementos que se quieren escribir, en este caso 1. El último argumento es el puntero al fichero en el que se quiere hacer la escritura.

A continuación, el programa escribe un array de 3 números enteros. En este caso la instrucción es:

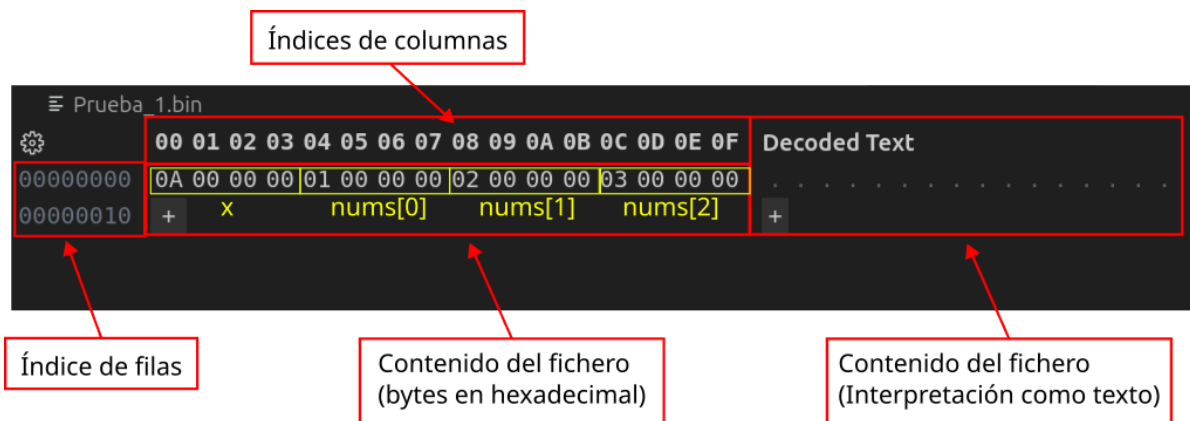
```
result = fwrite(nums, sizeof(int), 3, fid);
```

Ahora, el puntero a la variable que queremos escribir es el nombre del array que, como sabemos, es un puntero al primer elemento del array. El tamaño de los elementos que vamos a escribir es el de un número entero del tipo *int* y el número de elementos que queremos escribir es 3.

Si miras en la parte del explorador de archivos de VSCode, deberías ver que ha aparecido el fichero *Prueba_1.bin*. Si miras el fichero en el explorador de archivos o haciendo *DIR* en el terminal, verás que el tamaño del fichero es 16 bytes, que corresponden a 4 números *int* de 4 bytes cada uno. Pero no podrás abrir el fichero con un editor de texto, pues se trata de un fichero binario.

Si has instalado la extensión *Hex Editor* que se mencionó en el Apartado 1, puedes pulsar el botón derecho del ratón sobre el nombre del fichero y seleccionar *Abrir con...*, eligiendo a continuación *Hex Editor*. Deberías ver una imagen similar a la de la siguiente figura, donde se han indicado las distintas secciones para que puedas comprender su significado:

Puedes ver los 16 bytes del contenido del fichero, con su valor hexadecimal y con su interpretación co-



mo texto. El editor te muestra el contenido del fichero en una matriz de 16 columnas por las filas que sean necesarias, por necesidades de visualización, pero el fichero es una tira de bytes consecutivos, sin ninguna organización en filas o columnas.

La escritura en formato binario ofrece una ventaja importante cuando se trata de escribir estructuras. En el siguiente ejemplo, añadimos al fichero anterior una variable de tipo estructura. Para ello, el modo de apertura del fichero se ha establecido como “ab” (*append binary*). Además, tras ejecutar la instrucción de escritura en el fichero, se ha añadido una comprobación de que la escritura se hizo correctamente.

Ejemplo 4 Escritura de una estructura con comprobación de error

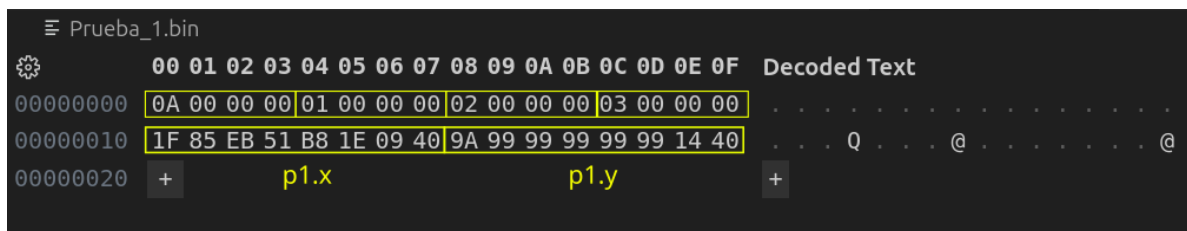
```
#include <stdio.h>

typedef struct {
    double x;
    double y;
} Punto;

int main() {

    // Abrir fichero
    FILE* fid = fopen("Prueba_1.bin", "ab");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }
    // Escribir en el fichero
    Punto p1 = {3.14, 5.15};
    int result = fwrite(&p1, sizeof(p1), 1, fid);
    // Comprobación de escritura correcta
    if (result != 1) {
        printf("Error al escribir en el archivo\n");
        return 1;
    }
    // Cerrar fichero
    fclose(fid);
    return 0;
}
```

En la siguiente figura puedes ver el contenido del fichero *Prueba_1.bin* abierto con el *Editor Hexadecimal*. Observa que el valor de los dos campos del *Punto p1* se han añadido al final del fichero.



10 ESCRITURA DE CARACTERES Y CADENAS EN FICHEROS BINARIOS

Aunque el formato del fichero sea binario, también se pueden guardar caracteres individuales o cadenas de caracteres. El siguiente código añade al fichero *Prueba_1.bin* un carácter individual y una cadena de caracteres completa:

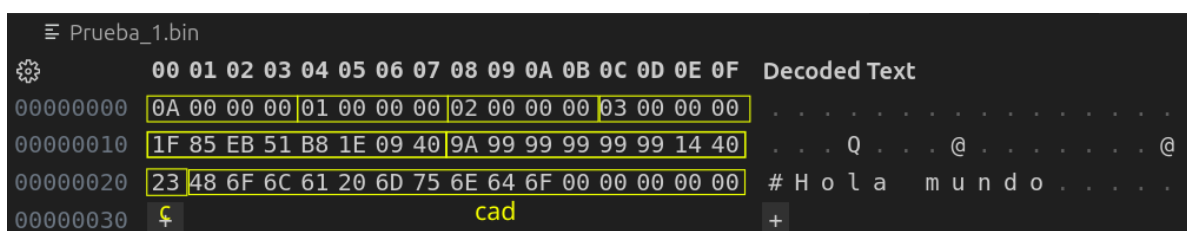
Ejemplo 5 Añadir un carácter y una cadena a *Prueba_1.bin*

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

int main() {

    // Abrir fichero
    FILE* fid = fopen("Prueba_1.bin", "ab");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }
    // Escribir 1 carácter en el fichero
    char c = '#';
    int result = fwrite(&c, sizeof(c), 1, fid);
    // Escribir una cadena completa en el fichero
    char cad[15] = "Hola mundo";
    result = fwrite(cad, sizeof(cad), 1, fid);
    // Cerrar fichero
    fclose(fid);
    return 0;
}
```

El contenido del fichero *Prueba_1.bin* en el editor:



Observa que la cadena *cad* se ha añadido entera, sus 15 bytes, incluida la parte que no tiene caracteres. También debes fijarte en que, ahora, la parte derecha del editor con la interpretación del contenido como texto, sí que muestra los caracteres.

Vamos a ver otro ejemplo para que comprendas mejor cómo se añaden arrays al fichero y, en concreto, cómo se pueden añadir arrays de caracteres.

Ejemplo 6 Añadir cuatro caracteres y una cadena a *Prueba_1.bin*

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

int main() {

    // Abrir fichero
    FILE* fid = fopen("Prueba_1.bin", "ab");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }

    char cad[15] = "Hola mundo";

    // Añadir 4 caracteres de la cadena
    int result = fwrite(cad, sizeof(char), 4, fid);

    // Añadir todos los caracteres válidos de la cadena
    result = fwrite(cad, strlen(cad), 1, fid);

    // Cerrar fichero
    fclose(fid);
    return 0;
}
```

El contenido del fichero:

	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000	0A 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
00000010	1F 85 EB 51 B8 1E 09 40 9A 99 99 99 99 14 40	. . . Q . . @ @
00000020	23 48 6F 6C 61 20 6D 75 6E 64 6F 00 00 00 00 00	# H o l a m u n d o
00000030	48 6F 6C 61 48 6F 6C 61 20 6D 75 6E 64 6F +	H o l a H o l a m u n d o +

Observa que se han ejecutado dos instrucciones *fwrite()*:

- En la primera, se añaden cuatro elementos, de tamaño 1 carácter.
- En la segunda, se añade un elemento de tamaño la longitud de la cadena.

Juega un poco con todos estos conceptos para comprender bien cómo funciona la instrucción *fwrite()* y cómo, la combinación del tamaño y el número de elementos, determina qué es lo que realmente se añade al fichero.

11 LECTURA DE FICHEROS BINARIOS

Para leer ficheros en formato binario se utiliza la instrucción *fread()*:

```
size_t fread(void* buffer, size_t n, size_t c, FILE* pFichero);
```

El funcionamiento, los parámetros y el valor devuelto son los mismos que los que hemos visto para la función *fwrite()* pero, en este caso, se leen elementos del fichero y se guardan en *buffer*, en vez de leerlos de *buffer* y guardarlos en el fichero como hacíamos en el Apartado 9.

Vamos a leer los datos del fichero *Prueba_1.bin* que generamos en los apartados anteriores:

Ejemplo 7 Lectura del fichero *Prueba_1.bin*

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

typedef struct {
    double x;
    double y;
} Punto;

int main() {
    // Abrir fichero
    FILE* fid = fopen("Prueba_1.bin", "rb");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }

    // Leer un número entero
    int n;
    fread(&n, sizeof(int), 1, fid);
    printf("n=%d\n", n);

    // Leer un array de 3 enteros
    int nums[3];
    fread(&nums, sizeof(int), 3, fid);
    printf("nums[0]=%d, nums[1]=%d, nums[2]=%d\n",
        nums[0], nums[1], nums[2]);

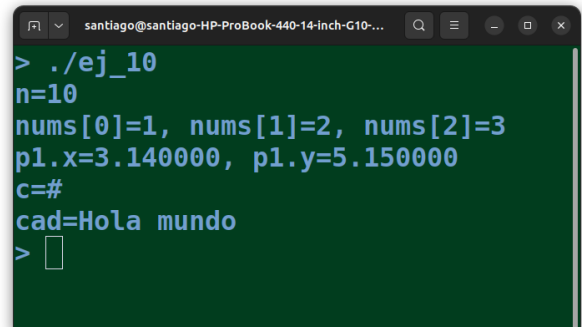
    // Leer un Punto
    Punto p1;
    fread(&p1, sizeof(Punto), 1, fid);
    printf("p1.x=%lf, p1.y=%lf\n", p1.x, p1.y);

    // Leer un carácter
    char c;
    fread(&c, sizeof(char), 1, fid);
    printf("c=%c\n", c);

    // Leer una cadena de tamaño 15
    char cad[15];
    fread(cad, sizeof(char), 15, fid);
    printf("cad=%s\n", cad);

    // Cerrar fichero
    fclose(fid);

    return 0;
}
```



```
santiago@santiago-HP-ProBook-440-14-inch-G10-...
> ./ej_10
n=10
nums[0]=1, nums[1]=2, nums[2]=3
p1.x=3.140000, p1.y=5.150000
c=#
cad=Hola mundo
>
```

Ficheros Binarios Vs Ficheros de Texto

Un inconveniente de los ficheros binarios es que, para poder leerlos, se necesita saber cómo fue grabado el fichero, si no no es posible leerlo. La simple inspección de su contenido no permite saber qué información contiene.

Es uno de los motivos por los que actualmente se tiende más a la utilización de ficheros de texto para guardar información. Los ficheros de texto es fácil abrirlos con un editor y organizar el programa de lectura, pues se ve con claridad cómo está organizada la información en el fichero.

No obstante, en este curso no estudiamos la lectura o escritura de ficheros de texto.

12 LA FUNCIÓN *feof()*. LECTURA DE FICHEROS CON BUCLES

Para poder hacer el ejemplo de este apartado, primero vamos a escribir un fichero con cinco elementos del tipo *Punto*, de forma que luego lo podamos utilizar para hacer la lectura del mismo.

Ejemplo 8 Creación del fichero *puntos.bin* con cinco puntos

```
#include <stdio.h>

typedef struct {
    double x;
    double y;
} Punto;

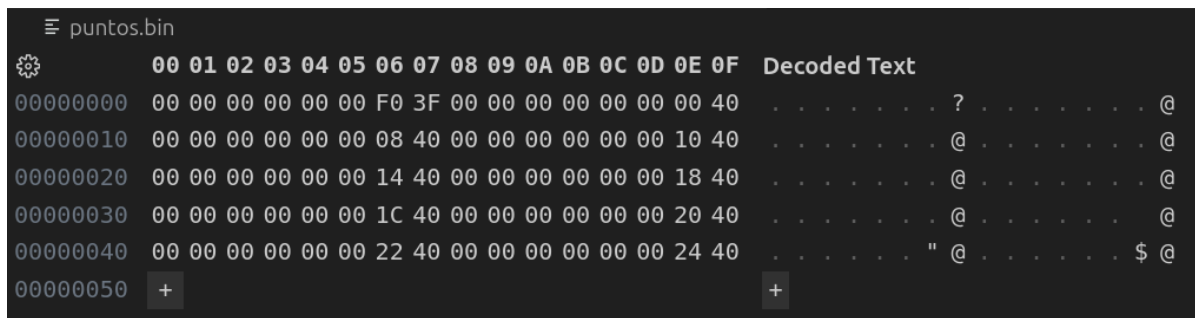
int main() {

    // Abrir fichero
    FILE* fid = fopen("puntos.bin", "wb");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }

    // Escribir cinco puntos
    Punto p1 = {1.0, 2.0};
    fwrite(&p1, sizeof(Punto), 1, fid);
    Punto p2 = {3.0, 4.0};
    fwrite(&p2, sizeof(Punto), 1, fid);
    Punto p3 = {5.0, 6.0};
    fwrite(&p3, sizeof(Punto), 1, fid);
    Punto p4 = {7.0, 8.0};
    fwrite(&p4, sizeof(Punto), 1, fid);
    Punto p5 = {9.0, 10.0};
    fwrite(&p5, sizeof(Punto), 1, fid);

    // Cerrar fichero
    fclose(fid);
    return 0;
}
```

La figura siguiente muestra el contenido del fichero *puntos.bin* que nos proporciona el editor hexadecimal.



Como no podía ser de otra manera, el fichero ocupa 80 bytes: 5 puntos \times 2 doubles \times 8 bytes.

Vamos a leer el fichero en el supuesto de que sabemos que contiene estructuras del tipo *Punto*, pero no sabemos cuántos puntos hay. Para ello, vamos a utilizar la función *feof()*.

El nombre de la función, *feof*, es el acrónimo de *File End Of File*. La signatura de la función es la siguiente:

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

La función recibe como argumento un puntero al *FILE* que se quiere comprobar y devuelve un cero, si no se ha llegado al final del fichero y otro número si ya se ha alcanzado el final.

Utilizando *feof()*, la lectura del fichero *puntos.bin* que se creó en el Ejemplo 8 se puede plantear como un bucle en el que se van leyendo puntos, uno a uno, hasta que se alcance el final del fichero, momento en el que se detendrá el proceso.

Para almacenar los puntos que se lean del fichero, el programa siguiente crea una base de datos basada en la técnica del array compactado que se ha estudiado en el curso. La estructura de la base de datos, que se ha llamado *Puntos* (en plural), tiene dos campos: un array de elementos *Punto* que puede albergar hasta 100 puntos y una variable entera llamada *numPuntos* que indica cuántos puntos contiene la base de datos.

Ejemplo 9 Lectura del fichero *puntos.bin*

```
#include <stdio.h>

typedef struct {
    double x;
    double y;
} Punto;

typedef struct {
    int numPuntos;
    Punto puntos[100];
} Puntos;

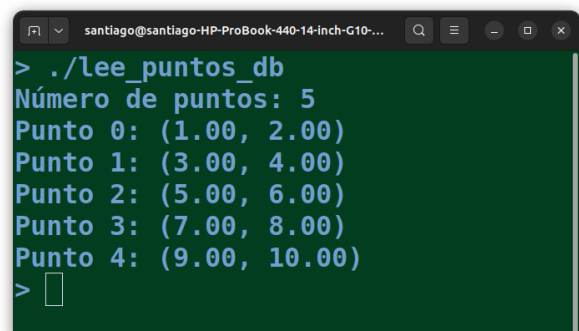
int main() {

    // Inicializar la base de datos
    Puntos db;
    db.numPuntos = 0;

    // Abrir fichero
    FILE* fid = fopen("puntos.bin", "rb");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }
    // Leer puntos
    while(feof(fid) == 0) {
        Punto p;
        int num_leidos =
            fread(&p, sizeof(Punto), 1, fid);
        if(num_leidos == 1) {
            db.puntos[db.numPuntos] = p;
            db.numPuntos++;
        }
    }
    // Cerrar fichero
    fclose(fid);

    // Mostrar resultado
    printf("Número de puntos: %d\n", db.numPuntos);
    for(int i=0; i< db.numPuntos; i++) {
        printf("Punto %d: (%.2lf, %.2lf)\n",
            i, db.puntos[i].x, db.puntos[i].y);
    }

    return 0;
}
```



```
santiago@santiago-HP-ProBook-440-14-inch-G10-...
> ./lee_puntos_db
Número de puntos: 5
Punto 0: (1.00, 2.00)
Punto 1: (3.00, 4.00)
Punto 2: (5.00, 6.00)
Punto 3: (7.00, 8.00)
Punto 4: (9.00, 10.00)
>
```

13 COMPROBACIÓN DE ERRORES DE LECTURA O ESCRITURA

Los procesos de lectura o escritura de información en ficheros son muy propensos a errores, por lo que conviene que los programas vayan comprobando que las operaciones se realizan de manera correcta.

Ya hemos visto que siempre, tras utilizar la instrucción *fopen()*, hay que comprobar que el fichero se ha abierto correctamente. También conviene comprobar que las operaciones de lectura o escritura se hacen correctamente.

En el Ejemplo 4 se vio cómo comprobar que la escritura de un elemento se ha realizado correctamente. En el Ejemplo 9 se comprobaba que la lectura de cada punto era correcta, antes de añadirlo a la base de datos.

Una comprobación que puede ser útil es la de que el número de elementos leídos en una operación *fread()* coincide con el número de elementos que se esperaba leer. Por ejemplo, en el siguiente código, se trata de leer el fichero *puntos.bin* que se creó en el Ejemplo 8 pero, en vez de utilizar un bucle controlado por *feof()*, como

se hizo en el Ejemplo 9, se tratan de leer 100 elementos. El valor devuelto por *fread()* nos indicará el número de puntos realmente leídos. Esta técnica puede servir de alternativa al uso del bucle y de *feof()* que se ha visto en el Ejemplo 9.

Ejemplo 10 Comprobación de puntos leídos de *puntos.bin*

```
#include <stdio.h>

typedef struct {
    double x;
    double y;
} Punto;

typedef struct {
    int numPuntos;
    Punto puntos[100];
} Puntos;

int main() {

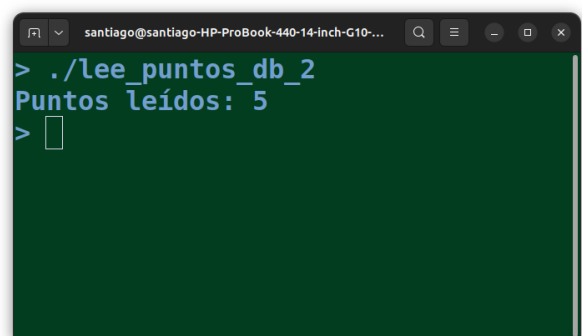
    // Inicializar base de datos
    Puntos db;
    db.numPuntos = 0;

    // Abrir fichero
    FILE* fid = fopen("puntos.bin", "rb");
    if (fid == NULL) {
        printf("Error al abrir el archivo\n");
        return 1;
    }
    // Leer puntos
    int num_leidos =
        fread(&db, sizeof(Punto), 100, fid);
    db.numPuntos = num_leidos;

    // Cerrar fichero
    fclose(fid);

    // Mostrar resultado
    printf("Puntos leídos: %d\n", num_leidos);

    return 0;
}
```



Este tipo de comprobaciones hay que hacerlo de manera regular. Además, el lenguaje C ofrece la función *ferror()* que permite comprobar si se ha producido algún error de cualquier tipo durante las operaciones de lectura o escritura:

```
int ferror(FILE* pFichero);
```

La función *ferror()* devuelve un cero si no ha detectado errores y otro número en caso contrario. Se pueden hacer las comprobaciones tras cada operación de lectura o escritura, tras un conjunto de ellas o antes de cerrar el fichero.