

Archivos binarios

INFO II – Ing. Federico Weber

Archivos de texto vs Archivos Binarios

- En C distinguimos entre archivos de texto y archivos binarios principalmente por cómo el sistema interpreta ciertos caracteres y cómo los datos son representados.
- Ya hemos manejado archivos de texto (conteniendo caracteres legibles, posiblemente formando líneas, números en formato de texto, etc.).
- Un archivo binario, en cambio, contiene datos en formato bruto (raw), tal cual los representa la memoria, sin traducción a texto legible.

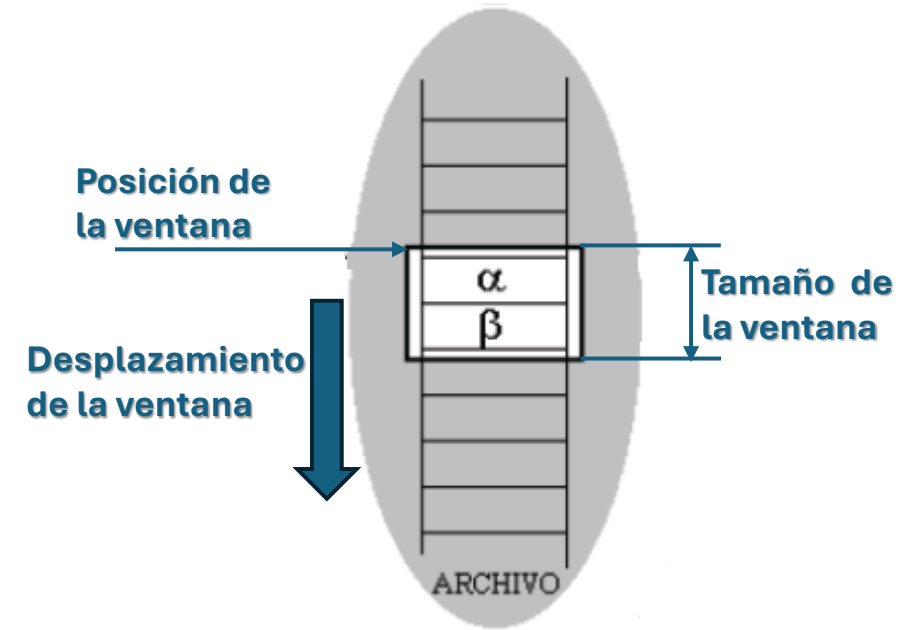
Modelo de la ventana (Ing. Jorge Argibay)

- Imaginemos un archivo como una fila muy larga de bytes.
- Cada byte ocupa una caja del mismo tamaño (1 byte).
- No se puede ver ni acceder al contenido completo a la vez.

Solo podemos interactuar con el archivo a través de una ventana.

- La ventana permite leer y escribir datos en el archivo.
- Todo acceso al archivo se hace a través de ella.
- Cada vez que se lee o escribe:
 - La ventana **se mueve automáticamente hacia adelante**
 - El movimiento es igual a la cantidad de bytes transferidos
 - Este desplazamiento es **automático e inevitable**
- Leer o escribir n bytes = cambiar el tamaño momentáneo de la ventana
- Mover la ventana a una posición específica = acceso directo
- Una vez movida, la lectura/escritura sigue siendo secuencial

La naturaleza intrínseca del manejo de archivos es secuencial. El acceso directo se logra a costa de reposicionar la ventana cada vez que se desea acceder a una posición determinada dentro del archivo.



A continuación veremos las funciones para escribir y leer n bytes:

- **fwrite**
- **fread**

Y para ubicar la ventana donde querramos:

- **fseek**
- **ftell**

Lectura y escritura en modo binario: **fread** y **fwrite**

- Cuando se trabaja con archivos binarios, a menudo es más conveniente usar las funciones **fread** y **fwrite**, definidas en **<stdio.h>**, que permiten leer/escribir bloques de bytes completos.
- Sus prototipos son:

size_t fread (void *ptr, size_t size, size_t count, FILE *f);

size_t fwrite (const void *ptr, size_t size, size_t count, FILE *f);

size_t

Es un tipo entero sin signo (unsigned) definido en **<stddef.h>** y otros encabezados estándar. Optimizado para representar tamaños de memoria y conteos de elementos.

void *ptr / const void *ptr

Un puntero genérico (puede apuntar a cualquier tipo de dato) . **void *** permite que las funciones trabajen con cualquier tipo de datos.

const en **fwrite()** indica que los datos originales no serán modificados.

Lectura y escritura en modo binario: **fread** y **fwrite**

- Sus prototipos son:

size_t fread (¹**void *ptr**, ²**size_t size**, ³**size_t count**, ⁴**FILE *fp**);

size_t fwrite (¹**const void *ptr**, ²**size_t size**, ³**size_t count**, ⁴**FILE *fp**);

Valor de retorno:

- El número de elementos leídos/escritos exitosamente.
- 0** en caso de error o que llego al fin de archivo.

Parámetros

- ptr**: Puntero al búfer de memoria donde se leerán/escribirán los datos
- size**: Tamaño en bytes de cada elemento a leer/escribir
 - Ejemplos:
 - sizeof(int)** para vectores de enteros
 - sizeof(double)** para vectores de números de punto flotante
 - sizeof(MiEstructura)** para vectores de estructuras personalizada
- count**: Número de elementos a leer/escribir (no bytes totales)

El total de bytes operados será (size * count)
- fp**: Puntero a la estructura FILE que representa el archivo
 - Obtenido previamente con `fopen()` en modo binario ("rb", "wb", etc.)

Uso de fwrite – ejemplo01.c

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

```
int main()
```

```
{
```

```
    int numeros[10] = {0,1,2,3,4,5,6,7,8,9};
```

```
    FILE *fp = fopen("datos.dat", "wb");
```

```
    if (fp == NULL) {
```

```
        perror("No se pudo abrir el archivo");
```

```
        return 1;
```

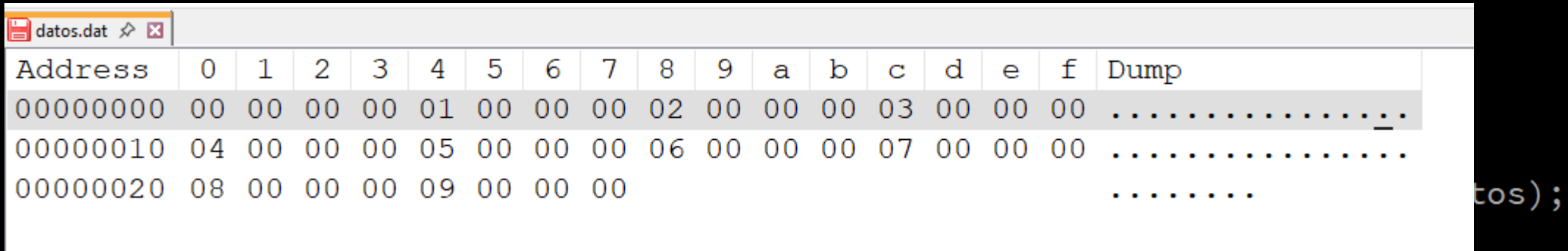
```
    }
```

```
    size_t escritos = fwrite (numeros, sizeof(int), 10, fp);
```

¿Cuántos bytes voy a escribir en el archivo?

40

→ ventana de escritura

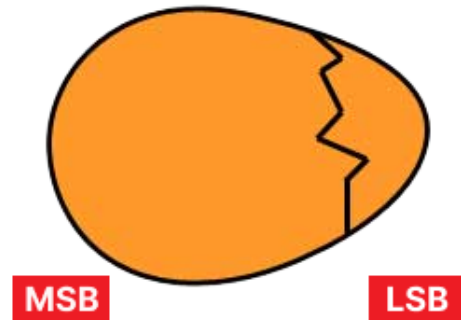


Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	00	00	00	00	01	00	00	00	02	00	00	00	03	00	00	00
00000010	04	00	00	00	05	00	00	00	06	00	00	00	07	00	00	00
00000020	08	00	00	00	09	00	00	00								

```
    return 0;
```

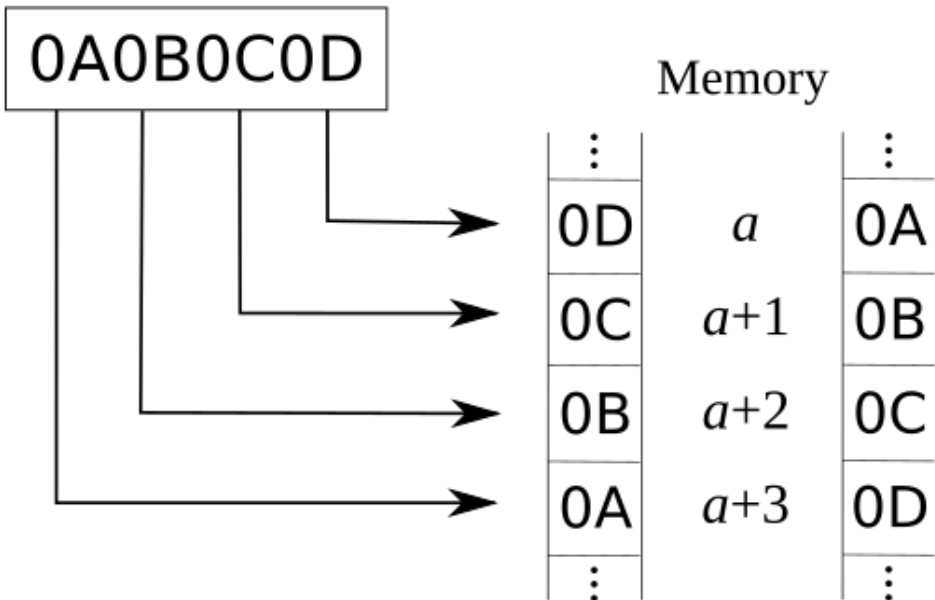
```
}
```


LITTLE ENDIAN - The Lilliputians break their eggs at the smaller end

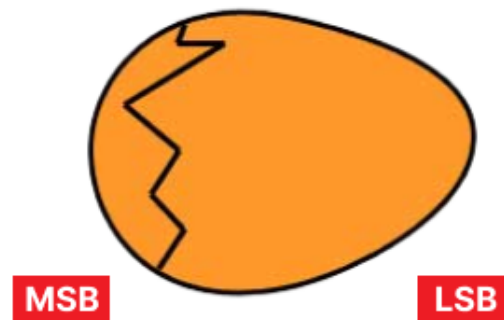


Little-endian

32-bit integer

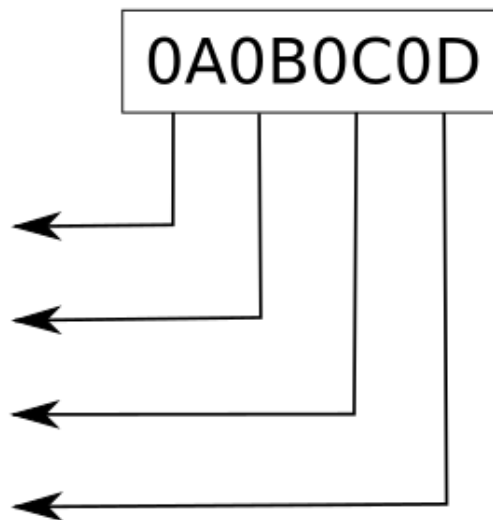


BIG ENDIAN - The Blefuscudians break their eggs at the big end.



Big-endian

32-bit integer



Little-Endian vs Big-Endian

- Jonathan Swift utilizó Big-Endians para describir a los blefuscudianos por su forma de cascar el huevo duro en su obra satírica *Los viajes de Gulliver* (1726).

Lo importante:

- La mayoría de las PCs actuales usan little-endian (como Intel y AMD).
- Algunos microcontroladores o arquitecturas "exóticas" usan big-endian (como algunos ARM o PowerPC).
- En red (protocolo IP, TCP), se usa **big-endian**, por eso se habla de "network byte order".

Uso de fread – ejemplo02.c

¿De a cuántos bytes leemos?

4

→ ventana de lectura

Cada vez que leo, la ventana se mueve automáticamente a la próxima posición.

Usando aritmética de punteros reacomodo el próximo lugar donde voy a guardar información.

¿Y si queremos leer los 10 enteros de una?

Solo “ajustamos el tamaño de la ventana”

```
int main()
{
    int numeros[10];
    int contador=0;
    int *ptr = numeros;

    FILE *fp = fopen("datos.dat", "rb");
    if (fp == NULL) { ...}

    while (fread (ptr, sizeof(int), 1, fp) == 1) {
        contador++;
        ptr++; }

    if (feof(fp))
        printf ("Fin de archivo correcto. %d enteros leidos\n", contador);
    else {
        perror ("Error en la lectura");
        fclose(fp);
        return 1; }

    printf ("\nImprimimos el contenido del archivo:\n");
    for (int i=0; i<contador; i++) {
        printf ("%d - ", numeros[i] ); }

    printf ("\n\nFin del programa.");
    fclose (fp);
    return 0;
}
```


Uso de fread – ejemplo03.c

¿De a cuántos bytes leemos?

40

→ ventana de lectura

Cada vez que leo, la ventana se mueve automáticamente a la próxima posición.

C:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe

Error en la lectura: No error
Press <RETURN> to close this window...

¿Qué paso acá?

¿Cómo funcionaba feof()?

feof() → solo se activa si se intentó leer después del fin del archivo.

```
int main()
{
    int numeros[10];

    FILE *fp = fopen("datos.dat", "rb");
    if (fp == NULL) { ... }

    fread (numeros, sizeof(int), 10, fp) ;

    if (feof(fp))
        printf ("Fin de archivo correcto.\n");
    else {
        perror ("Error en la lectura");
        fclose(fp);
        return 1;    }

    printf ("\nImprimimos el contenido del archivo:\n");
    for (int i=0; i<10; i++) {
        printf ("%d - ", numeros[i] ); }

    printf ("\n\nFin del programa.");
    fclose (fp);
    return 0;
}
```

Uso de fread – ejemplo03.c

Esta no es la forma mas usual de detectar si se termino de leer el archivo. Pero tampoco es usual querer leer un bloque único de 40 bytes.

La forma típica es hacer la lectura dentro de un **while** que evalúe el valor de retorno de **fread()**. Usar **feof()** fuera del lazo solo para confirmar que la lectura terminó limpiamente, y no por error. Tal cual el ejemplo02.c

```
int main()
{
    int numeros[10];

    FILE *fp = fopen("datos.dat", "rb");
    if (fp == NULL) { ... }

    size_t leidos = fread (numeros, sizeof(int), 10, fp) ;

    if (leidos < 10) {
        if (feof(fp)) {
            printf ("Fin de archivo correcto.\n"); }
        else {
            perror ("Error en la lectura");
            fclose(fp);
            return 1; } }
    else {
        printf ("Se leyeron los %d enteros correctamente", leidos);
    }

    printf ("\nImprimimos el contenido del archivo:\n");
    for (int i=0; i<10; i++) {
        printf ("%d - ", numeros[i] ); }

    printf ("\n\nFin del programa.");
    fclose (fp);
    return 0;
}
```

Diferencias clave entre archivos binarios y texto:

1. Representación:

- Un archivo de texto es aquel cuyo contenido está organizado en forma de caracteres imprimibles (generalmente ASCII o Unicode) agrupados en líneas (separadas por saltos de línea).
- Un archivo binario puede contener cualquier valor byte (0–255) en cualquier secuencia, representando estructuras de datos, números en formato interno, imágenes, audio, etc. Todo archivo en última instancia es una secuencia de bytes, pero en modo texto algunos bytes pueden interpretarse de forma especial (por ejemplo, `0x0A` como fin de línea).

Diferencias clave entre archivos binarios y texto:

2. Fin de línea (EOL):

- En modo texto, puede haber una conversión automática de caracteres de fin de línea. En sistemas Windows, al escribir un '\n' en modo texto, el sistema realmente grabará los dos caracteres 0x0D 0x0A (CR LF) en el archivo; de forma recíproca, al leer, convertirá ese CR LF de disco en un único '\n' en memoria.
- En Unix/Linux/macOS, el '\n' se representa internamente como 0x0A y usualmente no hay conversión (es el mismo valor en el archivo). En modo binario, no se realiza ninguna conversión de fin de línea: los bytes se leen y escriben tal cual son, preservando exactitud.
- Por ello, al trabajar con datos no-texto (ej. valores numéricos, estructuras) se debe usar modo binario, de lo contrario ciertos bytes podrían cambiar (por ejemplo, el byte 0x0A podría expandirse a 0x0D 0x0A en Windows).

Diferencias clave – ejemplo04.c

```
int main() {  
    FILE *ftxt, *fbin;  
  
    // Escribimos el byte 0x0A (newline) en modo texto  
    ftxt = fopen("modo_texto.txt", "w");  
    fputc(0x0A, ftxt); // mismo que fputc('\n', ftxt)  
    fclose(ftxt);  
  
    // Escribimos el byte 0x0A en modo binario  
    fbin = fopen("modo_binario.bin", "wb");  
    fputc(0x0A, fbin);  
    // char character = 0x0A;  
    // fwrite (&character, sizeof(char), 1, fbin);  
    fclose(fbin);  
  
    return 0;  
}
```

Abrimos con el
Notepad++

Diferencias clave

– ejemplo05.c

```
int main() {  
    FILE *ftxt, *fbin;  
    char c;  
    {  
        ...  
        // Leemos el archivo modo_texto.txt como binario  
        printf("Contenido de modo_texto.txt (leido como binario): ");  
        ftxt = fopen("modo_texto.txt", "rb");  
        while ((c = fgetc(ftxt)) != EOF) {  
            ;  
        }  
    }  
}
```

C:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe

```
Contenido de modo_texto.txt (leido como binario): 0D 0A  
Contenido de modo_binario.bin (leido como binario): 0A  
Press <RETURN> to close this window...
```

Al usar modo texto, el sistema puede cambiar lo que escribimos o leemos, especialmente los saltos de línea (\n). Por eso, cuando tratamos con datos binarios (estructuras, bytes exactos), debemos usar "wb" y "rb".

```
        // modo_binario.bin como binario  
        printf("Contenido de modo_binario.bin (leido como binario): ");  
        fbin = fopen("modo_binario.bin", "rb");  
        while ((c = fgetc(fbin)) != EOF) {  
            printf("%02X ", c);  
        }  
        // while (fread (&c, sizeof(char),1,fbin)) {  
        //     printf ("%02X ", c);  
        // }  
        fclose(fbin);  
        printf("\n");  
  
        return 0;  
    }  
}
```


Nota: Archivos de texto vs Archivos binarios

Importante:

- Al escribir datos binarios como enteros, floats, estructuras, etc., el archivo resultante dependerá de la representación interna en la máquina.
- Si luego se lee en una máquina con diferente endianness (orden de bytes) o tamaño de enteros, podría interpretarse de manera distinta.
- Esto no es un problema si el archivo solo será usado por programas similares o en el mismo entorno (por ejemplo, un log binario para depuración).
- Pero si la portabilidad del contenido es un requerimiento, suele ser preferible guardar en texto o definir un formato binario bien especificado.
 - ❖ Binario: ideal para velocidad, logs internos, y espacio cuando todo corre en el mismo entorno.
 - ❖ Texto (por ejemplo CSV): ideal cuando el archivo debe ser compartido, analizado, abierto en otros programas o portado entre sistemas.

El C/C++ son lenguajes de programación FUERTEMENTE TIPADOS

En esta tabla están los tipo INTEGER ESTÁNDAR definidos en C. Vean que el ESTÁNDAR solo especifica MINIMOS.

Tipo	Tamaño mínimo (bytes) según en standard C	Tamaño en (bytes) en este setup	Rango de valores mínimo garantizado
char	1	1	-128 a 127
unsigned char	1	1	0 a 255
short	2	2	-32768 a 32767
unsigned short	2	2	0 a 65535
int	2	4	-2147483648 a 2147483647 (-2B a 2B)
unsigned int	2	4	0 a 4294967295 (0 a 4B)
long	4	4	-2147483648 a 2147483647 (-2B a 2B)
unsigned long	4	4	0 a 4294967295 (0 a 4B)
long long	8	8	-9223372036854775808 a 9223372036854775807 (-9000T a 9000T)
unsigned long long	8	8	0 a 18446744073709551615 (0 a 18000T)

Tipos de tamaño fijo: <stdint.h>

- Los tipos tradicionales (int, long, etc.) cambian de tamaño según el compilador y la arquitectura.
- Para programas portables y con control exacto del tamaño (como en archivos binarios o protocolos), podemos usar los tipos de tamaño fijo.

Tipo	Tamaño mínimo (bytes) según en standard C	Rango de valores mínimo garantizado
int8_t	1	-128 a 127
uint8_t	1	0 a 255
int16_t	2	-32768 a 32767
uint16_t	2	0 a 65535
int32_t	4	-2147483648 a 2147483647 (-2B a 2B)
uint32_t	4	0 a 4294967295 (0 a 4B)
int64_t	8	-9223372036854775808 a 9223372036854775807 (-9000T a 9000T)
uint64_t	8	0 a 18446744073709551615 (0 a 18000T)

- Portabilidad total (funciona igual en cualquier plataforma)
- Ideal para archivos binarios, comunicación entre dispositivos, redes, etc.
- Recomendado para **sistemas embebidos**, firmware y aplicaciones críticas

Posicionamiento en el archivo: **fseek**

En ocasiones necesitamos movernos a una posición específica dentro de un archivo abierto (por ejemplo, saltar los primeros N bytes, o regresar al inicio, o leer el final). Para ello, C proporciona **fseek** y **ftell**:

- **int fseek (FILE *fp, long offset, int origen)**
 - Mueve la posición actual – *la ventana* – de lectura/escritura del archivo **fp**. El desplazamiento (**offset**) se interpreta relativo a **origen**, donde **origen** puede ser:
 - **SEEK_SET** – el inicio del archivo,
 - **SEEK_CUR** – la posición actual,
 - **SEEK_END** – el final del archivo.
 - Valor de retorno:
 - 0 si el desplazamiento fue exitoso.
 - != 0 (generalmente -1) si hubo un error (por ejemplo, mover más allá del inicio del archivo).

Por ejemplo:

- **fseek (fp, 0, SEEK_SET) →** vuelve al comienzo (offset 0 desde inicio)
- **fseek (fp, 10, SEEK_CUR) →** avanza 10 bytes desde donde esté
- **fseek (fp, -5, SEEK_END) →** se posiciona 5 bytes antes del final.

Mostrar el ultimo carácter de un archivo de texto - ejemplo07.c

Sin usar **fseek**, la única forma de llegar al ultimo carácter es leyendo primero TODO el archivo. Usando **fseek**, podemos acceder directamente.

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("archivo.txt", "r");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    // Mover el cursor al último byte del archivo
    fseek(fp, -1, SEEK_END);

    // Leer y mostrar el último carácter
    int c = fgetc(fp);
    printf("Último carácter: %c\n", c);

    fclose(fp);
    return 0;
}
```

Leer el 5º entero (posición 4)- ejemplo08.c

Leer el quinto número entero de un archivo binario que contiene 10 enteros. Sin **fseek()**, deberías leer y descartar los primeros 4.

Con **fseek()**, podés ir directo al que te interesa.

Antes de seguir:

Que los alumnos escriban la parte de código que falta acá, donde se cargan los 10 enteros tipo `uint32_t` en el archivo.

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

int main() {
    { ... }
    // Reabrimos para leer el 5º número (índice 4)
    fp = fopen("numeros.bin", "rb");
    if (!fp) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    // Saltar los primeros 4 números (4 * 4 bytes = 16 bytes)
    fseek(fp, 4 * sizeof(uint32_t), SEEK_SET);

    uint32_t numero;
    fread(&numero, sizeof(uint32_t), 1, fp);
    printf("El quinto numero es: %u\n", numero);

    fclose(fp);
    return 0;
}
```


editor con retroceso - ejemplo09.c

Que los alumnos digan si el programa funciona bien.

Depuración del código – Code Debugging -

- Vamos a ejecutar nuevamente este programa que graba caracteres y permite simular un "borrado" con '!'.
- Pero... tiene un pequeño error lógico que vamos a detectar usando un breakpoint, para ver cómo se comporta el programa paso a paso.
- ¿Qué es un **breakpoint**?
 - Un **breakpoint** o punto de interrupción es una marca que le decimos al programa: *“Detenete acá y mostrame qué está pasando”*.
 - Nos permite ver los valores de las variables, seguir el flujo línea por línea y encontrar errores más fácilmente.

Posicionamiento en el archivo: **ftell**

En ocasiones necesitamos movernos a una posición específica dentro de un archivo abierto (por ejemplo, saltar los primeros N bytes, o regresar al inicio, o leer el final). Para ello, C proporciona **fseek** y **ftell**:

- **long ftell (FILE *fp)**

- Devuelve la posición actual (en bytes) relativa al inicio del archivo.
- Esto permite saber en qué byte estamos. Útil, por ejemplo, para obtener el tamaño del archivo:
 - hacer **fseek(fp, 0, SEEK_END)**
 - luego usar **ftell(fp)** que dará el tamaño en bytes,
 - y luego opcionalmente volver al inicio con **fseek(fp, 0, SEEK_SET)**

Obtener el tamaño de un archivo – ejemplo10.c

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("imagen.png", "rb");
    if (!fp) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    fseek(fp, 0, SEEK_END);
    long size = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    printf("El archivo imagen.png pesa %ld bytes\n", size);

    fclose(fp);
    return 0;
}
```

Diferencias clave entre archivos binarios y texto:

3. Uso de **fseek/ftell**:

- En archivos de texto, debido a las conversiones, no todas las posiciones de bytes en el archivo físico corresponden linealmente a posiciones en el flujo lógico.
- La norma de C solo garantiza que **fseek** en modo texto funcione correctamente con desplazamientos obtenidos mediante **ftell** previamente, o con **fseek(f, 0, SEEK_SET)** al inicio.
- Es decir, en texto no es seguro hacer **fseek** arbitrarios (aunque en la práctica en muchos sistemas sí funciona, siempre es más predecible en binario). En archivos binarios, **fseek/ftell** operan de forma directa en bytes, lo que permite posicionarse en cualquier offset conocido del archivo.

Otras funciones: **ferror()**

La clase pasada (Archivos de texto: ejemplo10.c) leíamos un archivo carácter por carácter usando un **while** que terminaba cuando se alcanzaba **EOF**.

```
while ((c = fgetc(fp)) != EOF) {  
    putchar(c);  
}
```

Hasta aca todo bien. Pero después nos hicimos una pregunta:

¿Por qué salió el ciclo?

¿Porque terminamos de leer el archivo normalmente? ¿O porque hubo un error en la lectura?

En ese momento dijimos: “**si feof(fp)** es verdadero, entonces fue el fin del archivo. Si no, fue un error

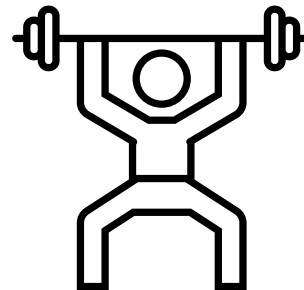
```
// Verificación real del fin de archivo  
if (feof(fp)) {  
    printf("\nFin del archivo alcanzado correctamente.\n");  
} else {  
    printf("\nLa lectura termino pero no por fin de archivo.\n");  
}
```

Y esto, es correcto, pero hay una forma directa de preguntar si hubo un error: **ferror(fp)**

Otras funciones: **ferror()**

- Es una función que verifica si se produjo un error en una operación de archivo (lectura o escritura) desde la última vez que se abrió o se limpió el error con **clearerr()** o **rewind()**. [notar que queda activo hasta que]
- **int ferror(FILE *fp);**
 - Devuelve 0 si no hubo error.
 - Devuelve distinto de 0 si ocurrió un error.
- ¿Por qué es necesaria?
 - Porque en muchas funciones como **fgetc**, **fread**, **fgets**, etc., no sabés si fallaron porque llegaron al final del archivo (**EOF**) o por un error real (por ejemplo, archivo dañado o disco lleno).
- **ferror()** te permite distinguir entre:
 - Fin del archivo → normal → **feof(fp)** es verdadero.
 - Error en lectura o escritura → anormal → **ferror(fp)** es verdadero.

Agregar esto al código le da mas robustez.
ejemplo12.c



Otras funciones:

- **rewind(fp)**
 - Hace lo mismo que **fseek (fp, 0, SEEK_SET)**
 - Pero además resetea a **ferror**
- **remove(fp)**
 - borrar archivos desde el programa.
- **rename("viejo.txt", "nuevo.txt")**

```
if (remove("datos.txt") == 0)
    printf("Archivo borrado con éxito.\n");
else
    perror("No se pudo borrar");
```

Ejercicios propuestos

1. Realizar un programa para copiar un archivo de texto en otro.
2. Realizar un programa que maneje un archivo de longitud desconocida, donde cada registro contiene el nombre de un alumno y cuatro notas. Hacerlo con un menú que permita crear el archivo, calcular el promedio, mostrarlo, buscar un registro determinado, modificar una nota, agregar registros y ordenar el archivo alfabéticamente. Realizar también el programa que genere el archivo y verificar que funcionen en conjunto.
3. Se tiene un archivo lista.dat que contiene la Base de Datos de artículos de un negocio:
 - Número de artículo(int),
 - Descripción (string de 30 caracteres),
 - precio (float),
 - proveedor(string de 30)
 - stock(int).

Actualizar esta Base de Datos aplicando un 20% de aumento a todos los artículos del proveedor Pérez.
Realizar también el programa que genere la base de datos y verificar que funcionen en conjunto.

An aerial photograph of a rugged coastline. The image shows dark, deep blue water with white foam from breaking waves. Several large, brownish-orange rock formations jutting out into the sea are visible. Some of these rocks are covered with patches of green vegetation. The overall scene is dramatic and scenic.

FIN DE LA CALSE