Capítulo 13 — Archivos

> *Capítulo intermedio/avanzado. Se apoya en lo aprendido sobre arreglos (`06`), punteros (`07`), `struct` (`08`), memoria dinámica (`09-memoria-dinamica`) y arenas (`12-arenas`). Se recomienda leerlo después de tener claros `malloc`, `free`, el uso de punteros y la idea de reservar memoria en bloques (arenas).*

13.1 ¿Por qué leer y escribir archivos?

Hasta ahora, nuestros programas intercambiaban información únicamente a través de la consola (entrada por teclado y salida por pantalla). Esto está bien para ejercicios cortos, pero en la vida real necesitamos **persistir datos**: guardar configuraciones, leer registros, procesar logs, exportar resultados o trabajar con formatos binarios como imágenes, audios o archivos compactados.

Un archivo nos permite guardar información en disco para usarla más tarde, compartirla entre programas o procesarla por lotes. El desafío es hacerlo de forma **clara, segura y correcta**.

13.1.1 Texto vs Binario

En C, los archivos se dividen en dos grandes categorías:

- **Archivos de texto**: almacenan caracteres legibles por humanos (ASCII/UTF-8). Cada línea suele terminar con un separador de fin de línea (`\n` o `\r\n` según el sistema). Ejemplos: `.txt`, `.csv`, `.json`, `.md`.
- **Archivos binarios**: almacenan datos tal como están en memoria (bytes crudos). No están pensados para leerse directamente con un editor de texto. Ejemplos: `.bmp`, `.png`, `.exe`, `.dat`.

La diferencia clave no es "que contengan texto", sino **cómo los interpretamos**. Leer un archivo binario como si fuera texto corrompe su estructura. Por eso, al abrir un archivo con `fopen()`, usamos modos distintos: `r`, `w`, `a` para texto y `rb`, `wb`, `ab` (más `+` cuando corresponde) para binario.

13.1.2 Recursos: abrir y cerrar

Todo archivo abierto consume un recurso del sistema. Por eso, **siempre hay que cerrarlo** cuando terminamos de usarlo. El par básico es:

```
fopen()  → abre el archivo (devuelve FILE*)
... leer/escribir ...
fclose() → cierra el archivo (libera recursos)
```

Un error común es olvidar `fclose()` o no cerrarlo en todos los caminos posibles (por ejemplo, si ocurre un error a mitad del proceso). Más adelante veremos cómo mitigar esto usando distintas estrategias de gestión de memoria/buffers.

13.2 Lectura y escritura de archivos de texto

La librería estándar de C (`stdio.h`) ofrece un conjunto de funciones para trabajar con archivos. Antes de usarlas, veamos **cada una**: cómo se llama, qué recibe y qué devuelve. Casi todas giran en torno al tipo `FILE *` , que es el **manejador** (handle) del archivo abierto.

13.2.1 Modos de apertura

`fopen` recibe la ruta y un **modo** que decide qué se puede hacer con el archivo:

Modo	Significado	¿Trunca el archivo?	¿Crea si no existe?
`"r"`	Solo lectura	No	No (falla si no existe)
`"w"`	Escritura	Sí (borra el contenido)	Sí
`"a"`	Agregar al final	No	Sí
`"r+"`	Lectura y escritura	No	No
`"w+"`	Lectura y escritura	Sí	Sí
`"a+"`	Lectura y escritura (escribe al final)	No	Sí

Agregando una `b` (`"rb"` , `"wb"` , `"ab"` , ...) se abre en **modo binario**.

> ***¿Texto o binario? ¿Qué cambia con el `\n`?*** *El **byte** `\n` (código `0x0A`) puede aparecer en cualquier archivo, sea de texto o binario: **no** es exclusivo del modo texto. Lo que cambia es cómo se **interpreta** al leer y escribir:*
>
> - *En **modo texto** (`"r"`, `"w"`), el sistema puede **traducir** los fin de línea. En Windows, al leer convierte la secuencia de dos bytes `\r\n` (CRLF, típica del disco) en un único `\n` en memoria; al escribir hace lo inverso. Además trata el byte `\x1A` (Ctrl+Z) como fin de archivo.*
> - *En **modo binario** (`"rb"`, `"wb"`) esa traducción **se desactiva**: cada byte entra y sale **tal cual**, sin ajustes.*
>
> *Esa es toda la diferencia (y solo importa en Windows; en Linux/macOS no hay traducción). Por eso los archivos cuyos bytes deben leerse **exactos** —imágenes, audio, o la cabecera ASCII de un PPM mezclada con píxeles— se abren con `"b"`. Ojo: el `\n` de la cabecera PPM del ejemplo (13.5) es un **byte real del archivo** (`0x0A`), no una traducción del modo texto.*

13.2.2 Referencia: funciones de texto

`fopen` — abrir un archivo

```
FILE *fopen(const char *path, const char *mode);
```

- **Recibe:** `path`, la ruta del archivo; `mode`, el modo de apertura (ver 13.2.1).
- **Devuelve:** un `FILE *` (el manejador del archivo abierto) o `NULL` si no se pudo abrir.
- **Ojo:** hay que verificar siempre `NULL` antes de usar el manejador.

`fclose` — cerrar un archivo

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

- **Recibe:** el `FILE *` a cerrar.
- **Devuelve:** `0` si salió bien, `EOF` si hubo un error al vaciar los buffers.
- **Ojo:** cerrar el archivo en **todos** los caminos del programa, incluso ante errores.

`fgetc` — leer un carácter

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

- **Recibe:** el `FILE *` del que se quiere leer.
- **Devuelve:** el carácter leído **como** `int`, o `EOF` al llegar al final o si hubo error.
- **Ojo:** el resultado se guarda en un `int`, no en un `char`: así el valor `EOF` no se confunde con un carácter válido.

`fputc` — escribir un carácter

```
int fputc(int c, FILE *stream);
```

- **Recibe:** `c`, el carácter a escribir (como `int`); `stream`, el archivo destino.
- **Devuelve:** el carácter escrito (como `int`), o `EOF` si falló.

`fgets` — leer una línea

```
char *fgets(char *buf, int n, FILE *stream);
```

- **Recibe:** `buf`, el buffer destino; `n`, su tamaño; `stream`, el archivo origen.
- **Devuelve:** `buf` si leyó algo, o `NULL` al llegar al final o si hubo error.
- **Ojo:** lee como máximo `n - 1` caracteres, o hasta encontrar `\n` (que **incluye**), o hasta el fin del archivo; y agrega el terminador `'\0'`.

`fputs` — escribir una cadena

```
int fputs(const char *str, FILE *stream);
```

- **Recibe:** `str`, la cadena a escribir; `stream`, el archivo destino.
- **Devuelve:** un valor no negativo si salió bien, `EOF` si falló.
- **Ojo: no** agrega `\n` automáticamente.

`fprintf` — escribir con formato

```
int fprintf(FILE *stream, const char *format, ...);
```

- **Recibe:** `stream`, el archivo destino; luego `format` y los argumentos, igual que `printf`.
- **Devuelve:** la cantidad de caracteres escritos, o un valor negativo si falló.

`fscanf` — leer con formato

```
int fscanf(FILE *stream, const char *format, ...);
```

- **Recibe:** `stream`, el archivo origen; luego `format` y los **punteros** destino, igual que `scanf`.
- **Devuelve:** la **cantidad de datos asignados con éxito** (no la de caracteres leídos), o `EOF` si el archivo terminó antes de la primera conversión.
- **Ojo:** por eso `fscanf(fp, "%d", &x)` "bien hecho" devuelve `1`, no el número leído. Detectar un fallo es comparar contra lo esperado (`!= 1`, `!= 3`, ...), no contra `EOF`.

Ejemplo. Supongamos que el archivo contiene esto (y que el cursor de lectura está al inicio):

```
80 60 255
```

y ejecutamos:

```
int width, height, maxval;
int n = fscanf(fp, "%d %d %d", &width, &height, &maxval);
```

- Cada `%d` lee **todos los dígitos** de un número y, además, **descarta los espacios que haya antes**. Por eso `"%d %d %d"` sirve tanto para `80 60 255` como si cada número estuviera en una línea distinta. El `&` (dirección de) es obligatorio: es dónde se guarda cada valor, **en orden** (`width` , `height` , `maxval`).
- El valor de retorno `n` sería `3` (se asignaron las tres variables). Con menos de tres números en el archivo, `n` sería menor; si el archivo terminara antes del primero, `n` sería `EOF` .

¿Cómo avanza el cursor? No avanza "un lugar por cada conversión". Avanza tantos **bytes como caracteres consumió** y se detiene en el primer carácter que **no** usó, que queda **sin leer**. Para verlo, ubiquemos cada carácter de `80 60 255\n` por su **posición** (`esp` = un espacio):

```
contenido: '8' '0' esp '6' '0' esp '2' '5' '5' '\n'
posición:   0   1   2   3   4   5   6   7   8   9
           └───┘ 1er %d
                 └─────────┘ 2º %d
                             └─────────────┘ 3er %d
```

Y leído paso a paso:

- **1er** `%d` → `width` **:** lee `"80"` y deja el cursor en la posición **2**.
- **2.º** `%d` → `height` **: salta** el espacio de la posición 2, lee `"60"` y queda en la **5**.
- **3.er** `%d` → `maxval` **: salta** el espacio de la posición 5, lee `"255"` y queda en la **9**.

> *Fijate en el gráfico: el cursor termina en la posición 9,* ***justo antes del*** `\n` *.* `fscanf` *consume todos los dígitos de cada número, pero* ***nunca*** *el* `\n` *que sigue al último: ese separador queda* ***sin leer****. Si tu programa necesita leer lo que viene inmediatamente después, tenés que "comerte" ese separador con* `fgetc()` *. Es exactamente lo que hace el ejemplo de la sección 13.5.2 antes de leer los píxeles.*

`feof` — ¿se llegó al final?

```c
int feof(FILE *stream);
```

- **Recibe:** el `FILE *`.
- **Devuelve:** un valor distinto de cero si ya se intentó leer más allá del final; `0` si no.
- **Ojo:** no anticipa si el próximo read va a fallar; recién es verdadero **después** de tocar el final.

`ferror` — ¿hubo error?

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

- **Recibe:** el `FILE *`.
- **Devuelve:** un valor distinto de cero si ocurrió un error de lectura/escritura; `0` si no.
- **Ojo:** permite distinguir un fin de archivo normal (`feof`) de un error real (`ferror`).

`perror` — describir el último error

```
void perror(const char *msg);
```

- **Recibe:** un mensaje propio (por ejemplo `"fopen"`).
- **Devuelve:** nada; imprime en `stderr` ese mensaje, seguido de `:` y el texto del último error del sistema. Es lo natural justo después de que `fopen` devuelve `NULL` .

`toupper` / `tolower` — cambiar mayúsculas

```
int toupper(int c);
int tolower(int c);
```

- **Reciben:** un carácter (como `int`).
- **Devuelven:** el mismo carácter convertido a mayúscula (o minúscula), o sin cambios si no es letra.
- **Requieren:** `#include <ctype.h>` .

> *`printf` , `scanf` , `malloc` , `free` y `sizeof` ya se vieron en capítulos anteriores; acá nos concentramos en lo específico de archivos.*

13.2.3 Ejemplo: leer un archivo carácter por carácter

Con `fgetc()` leés **un byte por vez**. El valor de retorno es un `int`, no un `char`, para que pueda distinguirse el valor `EOF` (fin de archivo, normalmente `-1`). Este es el patrón clásico de recorrido completo:

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

int main(void) {
    FILE *in  = fopen("entrada.txt", "r");
    FILE *out = fopen("salida.txt", "w");

    if (in == NULL || out == NULL) {
        perror("fopen");        /* imprime el motivo del fallo */
        return 1;
    }

    int ch;                     /* int, no char: debe poder guardar EOF */
    while ((ch = fgetc(in)) != EOF) {
        fputc(toupper(ch), out); /* ejemplo: pasa todo a mayúsculas */
    }

    fclose(out);
    fclose(in);
    return 0;
}
```

Observá tres detalles que se repiten en **todos** los programas con archivos:

1. Se verifica que `fopen()` no haya devuelto `NULL`.
2. El bucle corta cuando `fgetc()` devuelve `EOF`.
3. Se cierran **los dos** archivos, en orden inverso al que se abrieron.

13.2.4 Ejemplo: leer línea por línea con `fgets`

Para trabajar línea a línea, `fgets()` es más cómodo que `fgetc()`. Lee como máximo `n - 1` caracteres, o hasta encontrar un `\n` (que **deja incluido** en el buffer), o hasta el fin del archivo:

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("notas.txt", "r");
    if (fp == NULL) {
        perror("fopen");
        return 1;
    }

    char line[128];             /* buffer de tamaño fijo */
    int number = 1;
    while (fgets(line, sizeof line, fp) != NULL) {
        printf("%4d: %s", number, line);  /* line puede traer el '\n' */
        number++;
    }

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

> *Un buffer fijo obliga a elegir un tamaño máximo de línea. Si una línea es más larga que el buffer, `fgets()` la entrega **cortada** en varios pedazos. Para este capítulo, asumimos líneas razonables.*

13.2.5 Ejemplo: escribir con formato (`fprintf`)

`fprintf()` es a un archivo lo que `printf()` es a la pantalla: acepta el mismo formato.

```c
#include <stdio.h>

int main(void) {
    FILE *log = fopen("reporte.txt", "w");
    if (log == NULL) {
        perror("fopen");
        return 1;
    }

    int   workers = 3;
    float hours   = 7.5f;
    fprintf(log, "Trabajadores: %d\n", workers);
    fprintf(log, "Horas totales: %.1f\n", hours);

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

Con modo `"w"` el archivo se **trunca** (si ya existía, se borra su contenido). Para **agregar** al final sin borrar lo anterior, se usa el modo `"a"` .

13.3 Ejercicios: archivos de texto

> ***Regla fundamental:*** *ningún ejercicio incluye código de solución. Solo se describe el problema, entrada/salida, conceptos y criterios de aceptación. Se permiten fragmentos de entrada/salida esperada como ejemplos.*

Estos ejercicios trabajan sobre lo visto en la sección 13.2: `fgetc` , `fgets` , `fprintf` y `feof` .

Ejercicio 13.1 — Contador de líneas y caracteres (texto, básico)

Conceptos · `FILE*`, `fopen`, `fclose`, `fgetc`, `feof`

Contexto · En muchos editores o herramientas de análisis de texto necesitamos contar líneas y caracteres de un archivo. Esta es una simplificación: contar caracteres totales y cantidad de líneas de un archivo de texto plano.

Consigna · Escribir un programa que reciba por argumento (o por entrada) la ruta de un archivo de texto y muestre por pantalla:

- Cantidad total de caracteres (incluyendo saltos de línea).
- Cantidad total de líneas.

Entrada · Ruta de un archivo de texto plano. **Salida** · Dos líneas con los conteos, con formato claro y legible.

Ejemplo · Archivo `datos.txt` (corto a propósito):

```
Hola, mundo
C es genial
Fin
```

Cada línea termina en su `\n`, así que el archivo tiene **3** saltos de línea.

```
Entrada: datos.txt
Salida:  Caracteres: 28
         Líneas: 3
```

Identificadores sugeridos · `filePath`, `charCount`, `lineCount`, `fp`

Restricciones/pistas · Leer carácter por carácter con `fgetc()`. Detectar fin de archivo con `feof()`. Contar `\n` como salto de línea (no contar líneas vacías de forma especial, solo incrementar al encontrar `\n`). Siempre validar que el archivo se abrió correctamente.

Criterios de aceptación

- [ ] Abre el archivo en modo texto (`r`) y verifica `fopen != NULL` .
- [ ] Cuenta correctamente caracteres totales y líneas.
- [ ] Detecta correctamente fin de archivo.
- [ ] Cierra el archivo con `fclose()` antes de terminar.
- [ ] Muestra salida con formato claro.

Ejercicio 13.2 — Copia de archivo de texto (texto, básico)

Conceptos · `FILE*` , `fopen` , `fclose` , `fgetc` , `fputc`

Contexto · Copiar archivos es una operación fundamental (backup, exportar, procesar). Este ejercicio simplifica la copia byte-a-byte entre archivos de texto.

Consigna · Crear un programa que copie un archivo de texto `origen.txt` a `destino.txt` , preservando su contenido tal cual (carácter por carácter).

Entrada · Rutas de origen y destino (pueden pedirse por consola). **Salida** · Mensaje indicando si la copia fue exitosa o si hubo error.

Ejemplo · Archivo `origen.txt` :

```
Hola, mundo
C es genial
Fin
```

Después de correr el programa, `destino.txt` contiene **exactamente los mismos bytes**.

```
Entrada: origen.txt destino.txt
Salida:  Copia realizada con éxito.
```

Identificadores sugeridos · `src` , `dst` , `fpIn` , `fpOut` , `ch`

Restricciones/pistas · Abrir origen en modo `r` , destino en modo `w` . Leer con `fgetc()` y escribir con `fputc()` . Verificar aperturas y cierres. No modificar el contenido.

Criterios de aceptación

- [ ] Verifica que origen existe y se puede leer.
- [ ] Crea/escribe destino correctamente (trunca si existe).
- [ ] Copia todos los caracteres hasta EOF.
- [ ] Reporta errores de forma clara.
- [ ] Cierra ambos archivos.

Ejercicio 13.3 — Leer líneas y guardar filtradas (texto, intermedio)

Conceptos · `fgets`, buffers, cadenas, condicionales

Contexto · Muchas veces necesitamos filtrar líneas que cumplan cierto criterio (logs con un patrón, líneas que empiezan con una palabra, etc.). Simplificamos: leer un archivo de texto y guardar en otro **solo las líneas que contienen una palabra dada**.

Consigna · Dado un archivo de entrada, una palabra a buscar y un archivo de salida, copiar a salida únicamente aquellas líneas del archivo de entrada que **contengan** la palabra especificada (búsqueda case-sensitive simple).

Entrada · Ruta entrada, palabra a buscar, ruta salida. **Salida** · Archivo de salida con líneas filtradas. Mensaje con cantidad de líneas copiadas.

Ejemplo · Archivo `log.txt`:

```
[08:15] INFO  arranque del servidor
[08:16] ERROR no se pudo abrir config.txt
[08:17] INFO  escuchando en el puerto 8080
[08:18] ERROR conexión rechazada por el cliente
[08:19] WARN  reintento en 5 s
[08:20] ERROR timeout al esperar respuesta
```

Palabra a buscar: `ERROR` (case-sensitive). Debe quedar `errores.txt` con **solo** esas tres líneas, en el mismo orden:

```
[08:16] ERROR no se pudo abrir config.txt
[08:18] ERROR conexión rechazada por el cliente
[08:20] ERROR timeout al esperar respuesta
```

```
Entrada: log.txt ERROR errores.txt
Salida:  Se copiaron 3 líneas a errores.txt
```

Identificadores sugeridos · `inputPath`, `keyword`, `outputPath`, `lineBuf`, `count`

Restricciones/pistas · Usar `fgets()` con un buffer de tamaño razonable (fijo). No asumir líneas infinitamente largas. Buscar la palabra dentro de la línea (puede usarse recorrido simple). Abrir archivos en modo texto.

Criterios de aceptación

- [ ] Lee línea por línea con `fgets()`.
- [ ] Filtra correctamente líneas que contienen la palabra (coincidencia substring simple).
- [ ] Escribe solo líneas filtradas a archivo de salida.
- [ ] Informa cantidad de líneas copiadas.
- [ ] Maneja errores de apertura correctamente.

13.4 Lectura y escritura de archivos binarios

Cuando trabajamos con datos estructurados o con formatos como imágenes, necesitamos leer/escribir **bytes exactos**, sin que el sistema modifique nada (ni convierta finales de línea). Para eso usamos `fread()` y `fwrite()`, junto con funciones para ubicarse dentro del archivo.

> ***Antes de leer, mirá.*** *Un archivo binario no se puede abrir con un editor de texto: vas a ver basura, y encima el editor puede corromperlo. Para ver los bytes sin instalar nada, usá* ***ImHex Web*** *(https://web.imhex.werwolv.net/): un editor hexadecimal que corre entero en el navegador. Abrí ahí el BMP de la sección 13.8.6 o el PPM de la 13.5.1, ubicá los offsets y contrastá campo por campo lo que muestra la herramienta con lo que escribe tu programa.*

13.4.1 Referencia: funciones binarias

`fread` — leer un bloque

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

- **Recibe:** `ptr`, el buffer destino; `size`, el tamaño de **cada** elemento en bytes; `count`, cuántos elementos leer; `stream`, el archivo origen.
- **Devuelve:** la **cantidad de elementos** leídos (puede ser menor a `count` en el final o ante error). Si `size` es `1`, coincide con la cantidad de bytes leídos.
- **Ojo:** comparar el retorno con lo pedido para detectar lecturas incompletas; después, distinguir fin de archivo (`feof`) de error (`ferror`).

`fwrite` — escribir un bloque

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

- **Recibe:** los mismos parámetros que `fread`, pero `ptr` es lo que se **escribe**.
- **Devuelve:** la **cantidad de elementos** escritos (menor a `count` indica error).

`fseek` — mover el puntero de posición

```
int fseek(FILE *stream, long offset, int origin);
```

- **Recibe:** `offset`, el desplazamiento en bytes; `origin`, desde dónde contar: `SEEK_SET` (inicio), `SEEK_CUR` (posición actual) o `SEEK_END` (final).
- **Devuelve:** `0` si salió bien, distinto de cero si no (por ejemplo, un offset fuera del archivo).

`ftell` — posición actual

```
long ftell(FILE *stream);
```

- **Recibe:** el `FILE *`.
- **Devuelve:** la posición actual (offset en bytes desde el inicio) o `-1L` si hubo error.
- **Truco:** combinado con `fseek(fp, 0, SEEK_END)`, da el **tamaño** del archivo.

`rewind` — volver al inicio

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

- **Recibe:** el `FILE *`.
- **Devuelve:** nada; equivale a `fseek(fp, 0, SEEK_SET)` y además limpia el indicador de error.

> *`malloc` y `free` (para reservar el buffer de lectura) se vieron en el capítulo `09` (memoria dinámica). Recordá verificar el `NULL` que devuelve `malloc` y liberar con `free`.*

Con estas funciones leemos **bloques exactos de bytes**. Esto es fundamental para interpretar cabeceras de archivos binarios: leemos los primeros N bytes en un buffer o directamente en una estructura, y luego decidimos qué hacer según el contenido.

Importante: al trabajar con binario, el orden de bytes (*endianness*) puede importar dependiendo del formato y de la plataforma. En este capítulo nos enfocamos en **entender la estructura byte a byte del archivo** y leerlo correctamente según su especificación, sin asumir conversiones mágicas.

13.4.2 Ejemplo: averiguar el tamaño de un archivo

Antes de leer un archivo binario completo conviene saber **cuántos bytes tiene**. El truco es moverse al final con `fseek()`, preguntar la posición con `ftell()` y volver al principio con `rewind()`:

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("datos.bin", "rb");
    if (fp == NULL) {
        perror("fopen");
        return 1;
    }

    fseek(fp, 0, SEEK_END);   /* ir al final            */
    long size = ftell(fp);    /* posición == tamaño     */
    rewind(fp);               /* volver al principio    */

    printf("El archivo tiene %ld bytes\n", size);

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

> `long` *alcanza para archivos de tamaño normal. Para archivos gigantes existen* `fseeko()` *y* `ftello()` *(con* `off_t`*), pero no hacen falta en este capítulo.*

13.4.3 Ejemplo: leer el archivo completo a memoria

El patrón es: medir el tamaño, reservar un bloque con `malloc()`, leer todo de una sola vez con `fread()` y **siempre** liberar al terminar:

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

int main(void) {
    FILE *fp = fopen("datos.bin", "rb");
    if (fp == NULL) {
        perror("fopen");
        return 1;
    }

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

    unsigned char *buf = malloc((size_t)size);
    if (buf == NULL) {
        fprintf(stderr, "sin memoria\n");
        fclose(fp);
        return 1;
    }

    size_t read = fread(buf, 1, (size_t)size, fp);
    printf("Pedidos %ld bytes, leídos %zu\n", size, read);

    free(buf);        /* liberar SIEMPRE */
    fclose(fp);
    return 0;
}
```

> `fread()` *devuelve* ***cuántos elementos*** *leyó realmente. Puede leer menos de lo pedido si el archivo se acorta o si ocurre un error, así que conviene comparar con lo esperado.*

13.4.4 Ejemplo: leer una cabecera fija a una `struct`

Los archivos binarios suelen empezar con una cabecera de tamaño fijo. Se puede leer de una sola vez dentro de una `struct` que tenga **exactamente** el mismo diseño de bytes:

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

typedef struct {
    uint32_t magic;     /* 4 bytes */
    uint16_t version;   /* 2 bytes */
    uint16_t flags;     /* 2 bytes */
} FileHeader;

int main(void) {
    FILE *fp = fopen("archivo.bin", "rb");
    if (fp == NULL) {
        perror("fopen");
        return 1;
    }

    FileHeader header;
    if (fread(&header, sizeof header, 1, fp) != 1) {
        fprintf(stderr, "no se pudo leer la cabecera\n");
        fclose(fp);
        return 1;
    }

    printf("magic=0x%08X version=%u\n", header.magic, header.version);

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

> ***Cuidado con el padding:*** *el compilador puede insertar bytes de relleno dentro de una* `struct` *para alinear sus campos. Si el formato del archivo no tiene esos rellenos, el diseño de la* `struct` *en memoria puede* ***no coincidir*** *con el del archivo. En esos casos conviene leer los campos de a uno con* `fread()` *a variables sueltas, en lugar de a una* `struct` *entera.*

13.5 Ejemplo guiado: otro formato binario — PPM (Netpbm)

Antes del desafío de BMP, practiquemos con un formato binario **más simple todavía**: el **PPM** (Portable PixMap), uno de los formatos **Netpbm**. Su virtud didáctica es que su cabecera es **texto plano** y sus datos son una **matriz de colores sin compresión ni padding**, así que separa muy bien "leer la cabecera" de "leer los píxeles".

13.5.1 Descripción del formato PPM (variante P6)

Un archivo `.ppm` binario (variante **P6**) tiene esta estructura:

1. Los 2 bytes del **número mágico**: `P6` (ASCII).
2. Un carácter de espacio en blanco.
3. El **ancho** y el **alto** en píxeles, en ASCII decimal, separados por espacios.
4. El **valor máximo de componente** (por ejemplo `255`), en ASCII decimal.
5. Un **único** carácter de espacio en blanco.
6. Los **datos de píxeles**: para cada píxel, 3 bytes (`R`, `G`, `B`) **en ese orden**, recorriendo la imagen de **arriba hacia abajo** y de izquierda a derecha.

```
P6\n
<ancho> <alto>\n
<maxval>\n
<exactamente 3 * ancho * alto bytes, uno tras otro: R G B R G B ...>
```

Algunas diferencias con BMP que conviene notar:

	PPM (P6)	BMP (24 bpp)
Cabecera	Texto ASCII	Binaria
Orden de bytes por píxel	`R`, `G`, `B`	`B`, `G`, `R`
Orden de las filas	Arriba → abajo	Abajo → arriba
Relleno por fila (padding)	No tiene	Múltiplo de 4 bytes

13.5.2 Cómo leerlo

El siguiente programa lee la cabecera con `fscanf()` / `fgetc()` (son texto) y luego los píxeles con `fread()` (son binario). El truco clave está en el **paso 3**: después de `maxval` hay **exactamente un** carácter de espacio (habitualmente `\n`) que `fscanf()` **no** consume; hay

que comérselo con `fgetc()` antes de empezar a leer bytes, o el primer byte de datos se perdería.

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

int main(void) {
    FILE *fp = fopen("imagen.ppm", "rb");
    if (fp == NULL) {
        perror("fopen");
        return 1;
    }

    /* 1. número mágico: exactamente 2 caracteres */
    char magic[3] = {0};
    if (fscanf(fp, "%2s", magic) != 1 || magic[0] != 'P' || magic[1] != '6') {
        fprintf(stderr, "no es un PPM P6\n");
        fclose(fp);
        return 1;
    }

    /* 2. ancho, alto y valor máximo (son números en ASCII).
     *    Detalle de fscanf: ver la sección 13.2.2. */
    int width, height, maxval;
    if (fscanf(fp, "%d %d %d", &width, &height, &maxval) != 3) {
        fprintf(stderr, "cabecera PPM inválida\n");
        fclose(fp);
        return 1;
    }

    /* 3. consumir el ÚNICO blanco que sigue a maxval */
    fgetc(fp);

    /* 4. leer los píxeles: 3 bytes (R,G,B) por píxel, sin padding */
    size_t pixelCount = (size_t)width * (size_t)height;
    unsigned char *pixels = malloc(pixelCount * 3);
    if (pixels == NULL) {
        fprintf(stderr, "sin memoria\n");
        fclose(fp);
        return 1;
    }

    size_t read = fread(pixels, 3, pixelCount, fp);
    if (read != pixelCount) {
        fprintf(stderr, "datos incompletos\n");
        free(pixels);
        fclose(fp);
        return 1;
    }
```

```
    printf("PPM %dx%d, maxval %d\n", width, height, maxval);

    /* 5. mostrar el color del píxel central (fila/col del medio) */
    int cx = width  / 2;
    int cy = height / 2;
    size_t idx = ((size_t)cy * width + cx) * 3;
    printf("Píxel central: R=%u G=%u B=%u\n",
           pixels[idx + 0], pixels[idx + 1], pixels[idx + 2]);

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

*Este ejemplo es la **guía** para leer un mapa de píxeles. Fijate que las operaciones son las mismas que necesitarás para BMP: leer y validar la firma, extraer dimensiones, consumir el separador exacto entre cabecera y datos, y mapear `(x, y)` a un índice del buffer. La diferencia es que BMP agrega **orden invertido de filas** y **relleno (padding) por fila**, que veremos en la sección 13.8.*

13.6 Gestión de memoria: `malloc/free` vs Arenas

Al leer archivos, especialmente archivos binarios de tamaño variable o con estructuras anidadas, necesitamos almacenar lo leído en memoria. Tenemos dos enfoques principales:

13.6.1 Con `malloc` y `free`

- Reservamos exactamente lo que necesitamos (`malloc` del tamaño leído).
- Liberamos cada bloque cuando ya no lo usamos (`free`).
- **Ventaja:** memoria justa, se devuelve al sistema cuando corresponde.
- **Desventaja:** si leemos muchos bloques o estructuras pequeñas, debemos rastrear cada `free` . También complica liberar todo ante un error a mitad del proceso.

13.6.2 Con arenas (bump allocator)

Recordando el concepto del Capítulo 12: reservamos **un bloque grande** una sola vez, avanzamos un cursor para repartir memoria y **liberamos todo junto** (reiniciamos cursor o hacemos un único `free` del bloque de arena).

Aplicado a lectura de archivos:

- Podemos leer todo el archivo a un buffer grande en memoria (o leer en chunks) y repartir sub-buffers desde una arena, **sin liberar elemento por elemento**.
- Útil cuando procesamos un archivo completo en una pasada (o varias), pero no necesitamos conservar datos individuales mucho tiempo después de procesarlos.
- Reduce riesgo de *leaks*, *double-free* y simplifica el manejo de errores.

Nota conceptual: ambos enfoques son válidos. El objetivo de este capítulo es que aprendas a **leer archivos correctamente** y que puedas implementar la lectura **usando malloc/free** y también **usando arenas**, entendiendo cuándo conviene cada uno.

13.7 Errores y robustez

Al trabajar con archivos, hay que considerar fallas posibles:

- El archivo no existe o no tenemos permisos (`fopen` devuelve `NULL`).
- Fallo al leer/escribir (`ferror` indica error, funciones devuelven valores especiales como `EOF`).
- Fin de archivo inesperado (`feof`).
- Archivo más corto o más largo de lo esperado.
- Lectura binaria con offsets incorrectos (podemos leer fuera de lo que realmente hay).

Buenas prácticas:

1. Verificar siempre el resultado de `fopen()` .
2. Comprobar retorno de `fread/fwrite` (cantidad de elementos leídos/escritos). Si es menor a lo esperado, chequear `feof` / `ferror` .
3. No asumir tamaños: validar cabeceras antes de leer más datos.
4. Cerrar el archivo en todos los caminos (incluso ante errores). Esto puede hacerse con cuidado o estructurando el código para asegurar el cierre.

13.8 El formato BMP: documentación de referencia

Antes del desafío final, necesitamos entender el formato que vamos a interpretar: **BMP** (Windows Bitmap). Lo elegimos porque es uno de los formatos de imagen **más simples** de leer: guarda los píxeles **sin compresión** y documenta su estructura con cabeceras bien definidas.

13.8.1 Estructura general de un BMP

Un archivo `.bmp` se compone de estas partes, en este orden:

1. **BITMAPFILEHEADER** (14 bytes): firma y metadatos del archivo.
2. **BITMAPINFOHEADER** (40 bytes en la versión más común) u otra variante: describe la imagen.
3. **Paleta de colores** (opcional): solo para profundidades de 1, 4 u 8 bits.
4. **Datos de píxeles** (pixel data): la matriz de píxeles.

13.8.2 BITMAPFILEHEADER (14 bytes)

Offset	Tamaño	Campo	Descripción
0	2	`bfType`	Firma: siempre los bytes `0x42 0x4D` (ASCII "BM").
2	4	`bfSize`	Tamaño total del archivo en bytes (`uint32_t`).
6	2	`bfReserved1`	Reservado, debe ser 0.
8	2	`bfReserved2`	Reservado, debe ser 0.
10	4	`bfOffBits`	Offset en bytes donde comienzan los datos de píxeles.

13.8.3 BITMAPINFOHEADER (40 bytes, versión más común)

Offset	Tamaño	Campo	Descripción
14	4	`biSize`	Tamaño de esta cabecera: 40 bytes.
18	4	`biWidth`	Ancho de la imagen en píxeles (signed, puede ser negativo = top-down).
22	4	`biHeight`	Alto en píxeles (signed; si es negativo, filas de arriba a abajo).
26	2	`biPlanes`	Número de planos de color. Siempre 1.
28	2	`biBitCount`	Bits por píxel: 1, 4, 8, 16, 24 o 32.
30	4	`biCompression`	Tipo de compresión: 0 = BI_RGB (sin compresión).
34	4	`biSizeImage`	Tamaño de los datos de píxeles (0 si sin compresión).
38	4	`biXPelsPerMeter`	Resolución horizontal (puede ignorarse).
42	4	`biYPelsPerMeter`	Resolución vertical (puede ignorarse).
46	4	`biClrUsed`	Número de colores usados (0 = usar todos).
50	4	`biClrImportant`	Colores importantes (0 = todos importantes).

Resumen: desde el offset 14 hasta el 53 (40 bytes) está toda la información que necesitamos para saber cómo leer los píxeles.

13.8.4 Datos de píxeles

Los píxeles comienzan en el offset indicado por `bfOffBits` (campo del archivo, offset 10).

Para **24 bits por píxel** (el caso más simple para empezar):

- Cada píxel ocupa **3 bytes**: `B`, `G`, `R` (azul, verde, rojo) **en ese orden**. Es decir, el byte 0 es el azul, el byte 1 el verde y el byte 2 el rojo.
- Cada **fila** de píxeles se guarda de **abajo hacia arriba** en BMP (bottom-up): la primera fila del archivo es la fila inferior de la imagen, salvo que `biHeight` sea negativo (top-down).
- Cada fila debe ocupar un múltiplo de **4 bytes** (alineación DWORD). Si el ancho en bytes no es múltiplo de 4, se agregan **bytes de relleno (padding)** al final de cada fila.

La cantidad de bytes por fila (sin padding) es:

```
bytesPorFila = biWidth * 3   // para 24 bits
// luego se redondea hacia arriba a múltiplo de 4:
bytesPorFila = ((biWidth * 3 + 3) / 4) * 4
```

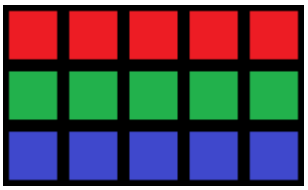
Las filas reales se leen desde el final hacia el principio si la imagen es bottom-up.

13.8.5 Paleta (solo para 1, 4 u 8 bpp)

Para 8 bits por píxel, cada píxel es 1 byte que es un **índice** a una tabla de 256 colores (paleta). Cada entrada de la paleta ocupa 4 bytes: `B`, `G`, `R`, reservado (0). La paleta comienza inmediatamente después de la cabecera de bits (offset `14 + biSize`).

13.8.6 Anatomía de un BMP real: ejemplo byte a byte

Las tablas de arriba describen el formato en abstracto. Ahora veamos un **BMP real**, chiquito a propósito (5×3 píxeles, 102 bytes), para seguir cada campo sobre los bytes mismos. Es el mismo archivo que usamos más adelante para verificar el Ejercicio 13.8.



Las filas del render son, de arriba a abajo: **rojo**, **verde**, **azul** — en sentido inverso a cómo están guardadas en el archivo, porque el BMP apunta hacia arriba (mirá "El engaño de bottom-up"). Las **líneas negras** son solo la grilla: marcan dónde termina un píxel y empieza el siguiente, y no existen en el archivo (son un agregado del render). Ahora sus bytes:

```
Offset  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x0000  42 4D 66 00 00 00 00 00 00 00 36 00 00 00 28 00
0x0010  00 00 05 00 00 00 03 00 00 00 01 00 18 00 00 00
0x0020  00 00 30 00 00 00 C3 0E 00 00 C3 0E 00 00 00 00
0x0030  00 00 00 00 00 00 CC 48 3F CC 48 3F CC 48 3F CC
0x0040  48 3F CC 48 3F 00 4C B1 22 4C B1 22 4C B1 22 4C
0x0050  B1 22 4C B1 22 00 24 1C ED 24 1C ED 24 1C ED 24
0x0060  1C ED 24 1C ED 00
```

Los 54 bytes de cabecera

Todos los enteros están en **little-endian**: el byte menos significativo va primero. Por eso `66 00 00 00` es `102`, y no un número gigante:

Offset	Bytes	Campo	Valor	Qué significa
0	`42 4D`	`bfType`	—	Firma "BM" (ASCII). Si no empieza así, no es un BMP.
2	`66 00 00 00`	`bfSize`	102	Tamaño total del archivo.
6	`00 00 00 00`	reservados	0	Ignorar.
10	`36 00 00 00`	`bfOffBits`	54	Los píxeles arrancan en el byte 54.
14	`28 00 00 00`	`biSize`	40	Cabecera de bits de 40 bytes.
18	`05 00 00 00`	`biWidth`	5	Ancho: 5 píxeles.
22	`03 00 00 00`	`biHeight`	3	Alto: 3 píxeles. **Positivo = bottom-up**.
26	`01 00`	`biPlanes`	1	Siempre 1.
28	`18 00`	`biBitCount`	24	24 bits por píxel = 3 bytes por píxel.
30	`00 00 00 00`	`biCompression`	0	`BI_RGB`: sin compresión.
34	`30 00 00 00`	`biSizeImage`	48	Datos de píxeles: 3 filas × 16 bytes.
38	`C3 0E 00 00`	`biXPelsPerMeter`	3779	Resolución; se puede ignorar.
42	`C3 0E 00 00`	`biYPelsPerMeter`	3779	Resolución; se puede ignorar.
46	`00 00 00 00`	`biClrUsed`	0	Sin paleta (24 bpp no usa paleta).
50	`00 00 00 00`	`biClrImportant`	0	Todos importantes.

Los 48 bytes de píxeles (con padding)

Cada píxel son **3 bytes en orden** `B` , `G` , `R` —azul, verde, rojo—, es decir **al revés del RGB** que suele usarse. Una fila de 5 píxeles son 15 bytes, pero BMP alinea cada fila a **4 bytes**: 15 se redondea hacia arriba a **16**. Esos 48 bytes quedan así:

Fila del archivo	Bytes	Color resultante	Relleno
1 (offset 54)	`CC 48 3F` × 5	azul · `RGB(63, 72, 204)`	`00`
2 (offset 70)	`4C B1 22` × 5	verde · `RGB(34, 177, 76)`	`00`
3 (offset 86)	`24 1C ED` × 5	rojo · `RGB(237, 28, 36)`	`00`

Fijate en los offsets: 54 + 16 = 70, 70 + 16 = 86, 86 + 16 = **102**, que es exactamente el `bfSize` de la cabecera. Todo cierra. El byte de relleno es `0x00` acá, pero **no está garantizado**: el contenido del padding es indefinido, así que lo único que podés hacer con él es **saltarlo**.

El engaño de bottom-up

`biHeight` es `3` (**positivo**), así que las filas están guardadas **de abajo hacia arriba**: la **primera** fila del archivo es la **inferior** de la imagen.

Orden en el archivo	Orden en pantalla
fila 1 → azul	fila 1 → rojo
fila 2 → verde	fila 2 → verde
fila 3 → rojo	fila 3 → azul

Si ignorás esto, la imagen sale **dada vuelta verticalmente**. Para corregirlo, al cargar el buffer copiá la fila `i` del archivo a la fila `height - 1 - i` de la imagen.

> ***Check rápido.*** *Con estos datos a mano, el píxel* ***central*** *de la imagen es el de la coordenada (2, 1) — columna 2, fila 1— y es* ***verde****:* `RGB(34, 177, 76)` *. Es una forma rapidísima de saber si tu programa leyó bien las cabeceras y el padding.*

13.8.7 Cómo leer un BMP (flujo sugerido)

1. Abrir el archivo en binario (`rb`).
2. Leer los 14 bytes del **BITMAPFILEHEADER**.
3. Verificar que `bfType == 0x4D42` (los 2 bytes; al leer como `uint16_t` little-endian da 0x4D42).
4. Leer los siguientes bytes según `biSize` para armar la cabecera de bits (40 bytes típicos).
5. Verificar `biCompression == 0` y que `biBitCount` sea manejable (recomendado 24).
6. Calcular `bytesPorFila` (con alineación a 4 bytes).
7. Posicionarse en `bfOffBits` y leer la matriz de píxeles.
8. Interpretar cada píxel (3 bytes: B, G, R para 24 bpp; índice a paleta para 8 bpp).
9. Mostrar o procesar los colores.

> *Los enteros de 4 bytes (como `bfSize` , `biWidth` , `biHeight`) se guardan en **little-endian**: el byte menos significativo va primero. Leerlos con `fread` en un `uint32_t` funciona bien en máquinas little-endian (x86/x64).*

13.9 Mostrar la imagen en pantalla (opcional): SDL2

Hasta acá todo el trabajo fue **leer e interpretar** bytes. Un objetivo opcional y muy motivador es **mostrar la imagen** en una ventana. C no trae esto en su biblioteca estándar: dibujar ventanas es una tarea del sistema operativo, y por eso se usa una biblioteca externa. Una de las más simples para empezar es **SDL2**.

> *Esta sección es **opcional** y va más allá del objetivo central del capítulo (leer archivos). SDL2 es una **dependencia externa**; si tu entorno no la tiene, podés saltearla sin problema.*

13.9.1 La idea

Ya tenés los píxeles en memoria, con 3 bytes por píxel. SDL2 ofrece una función que **envuelve un buffer de píxeles** en una superficie dibujable, sin copiar los datos:

```
SDL_Surface *surf = SDL_CreateRGBSurfaceFrom(
        pixels,      /* puntero a los datos      */
        width,       /* ancho en píxeles         */
        height,      /* alto en píxeles          */
        24,          /* bits por píxel           */
        width * 3,   /* pitch: bytes por fila    */
        rmask, gmask, bmask, 0);
```

Las **máscaras** (`rmask` , `gmask` , `bmask`) le dicen a SDL cómo están ordenados los bytes de color dentro de cada píxel. Si tu buffer está en orden `R, G, B` (como el que produjimos al leer el PPM, o al normalizar un BMP), en una máquina little-endian las máscaras son:

```
rmask = 0x000000FF
gmask = 0x0000FF00
bmask = 0x00FF0000
```

Si en cambio guardaste los píxeles en el orden `B, G, R` tal cual vienen en BMP, las máscaras se **intercambian** entre rojo y azul. Este es el error más común: colores correctos pero con rojo y azul cambiados.

13.9.2 Programa mínimo de visualización

Este programa asume que ya cargaste la imagen en `pixels` (un buffer `width * height * 3` en orden `R, G, B`, con la fila `0` **arriba**). Muestra la ventana hasta que la cierres o presiones una tecla:

```c
#include <SDL2/SDL.h>

/* Se asume que ya existen:  unsigned char *pixels;  int width, height; */

int main(void) {
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        SDL_Log("SDL_Init falló: %s", SDL_GetError());
        return 1;
    }

    SDL_Window *win = SDL_CreateWindow("Imagen",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        width, height, SDL_WINDOW_SHOWN);

    SDL_Surface *surf = SDL_CreateRGBSurfaceFrom(
        pixels, width, height, 24, width * 3,
        0x000000FF, 0x0000FF00, 0x00FF0000, 0);

    int running = 1;
    SDL_Event event;
    while (running) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT ||
               (event.type == SDL_KEYDOWN && event.key.keysym.sym == SDLK_ESCAPE)) {
                running = 0;
            }
        }

        SDL_Surface *screen = SDL_GetWindowSurface(win);
        SDL_BlitSurface(surf, NULL, screen, NULL);
        SDL_UpdateWindowSurface(win);
    }

    SDL_FreeSurface(surf);
    SDL_DestroyWindow(win);
    SDL_Quit();
    return 0;
}
```

13.9.3 El detalle de las filas: BMP es bottom-up

SDL espera la **fila 0 arriba**. BMP, en cambio, guarda la imagen **de abajo hacia arriba**. Si volcás los píxeles tal como los leíste, la imagen va a aparecer **invertida verticalmente**. La solución es copiar cada fila a su posición espejada (`fila destino = height - 1 - fila origen`) al momento de cargar el buffer.

13.9.4 Compilar y enlazar

SDL2 se enlaza como cualquier biblioteca externa. Con `gcc` y la utilidad `sdl2-config` :

```
gcc mostrar.c -o mostrar $(sdl2-config --cflags --libs)
```

En Windows con MinGW/MSYS2, instalá el paquete `mingw-w64-x86_64-SDL2` y compilá con `-lSDL2` (además hay que enlazar el subsistema de consola `-mwindows` o `-lmingw32` según el caso). En distribuciones Linux, el paquete suele llamarse `libsdl2-dev` .

> *Regla general de este libro: **no** incorporar dependencias externas salvo que sean el tema explícito del capítulo. Esta sección es la excepción deliberada y por eso queda marcada como opcional.*

13.10 Ejercicios: archivos binarios

Estos ejercicios ponen a prueba la segunda mitad del capítulo: archivos binarios con `fread` y `fwrite` (sección 13.4), el formato PPM (13.5), arenas (13.6), errores (13.7) y el formato BMP (13.8). La regla fundamental de la sección 13.3 sigue en pie: **ningún ejercicio incluye código de solución**.

Ejercicio 13.4 — Lectura binaria con `fread` (binario, intermedio)

Conceptos · Archivos binarios, `fopen` con `rb` , `fread` , buffers de bytes

Contexto · Necesitamos leer un archivo binario completo para analizar su contenido byte a byte. Este ejercicio es preparatorio para interpretar formatos estructurados.

Consigna · Leer un archivo binario completo en memoria (usando `malloc`) y mostrar:

- Tamaño total en bytes (puede estimarse leyendo hasta EOF o con `fseek` / `ftell`).
- Los primeros `N` bytes en formato hexadecimal (hex dump simplificado), donde `N` se pide como parámetro (o se fija, p.ej. 32).

Entrada · Ruta del archivo binario y valor `N` (cantidad de bytes a mostrar). **Salida** · Tamaño total y listado hex/simple de primeros N bytes.

Ejemplo

```
Entrada: datos.bin 16
Salida:  Tamaño total: 256 bytes
         00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
```

Identificadores sugeridos · `path`, `n`, `fp`, `buf`, `size`, `bytesRead`

Restricciones/pistas · Abrir con modo `rb`. Para obtener tamaño total: se puede hacer `fseek(fp, 0, SEEK_END)`, `ftell(fp)`, `rewind(fp)`. Luego reservar buffer con `malloc(size)` y leer todo con `fread(buf, 1, size, fp)`. Verificar que `fread` leyó lo esperado. **No** liberar de forma incorrecta (recordar `free(buf)` al final).

Criterios de aceptación

- [ ] Abre archivo binario correctamente (`rb`).
- [ ] Calcula tamaño total de forma segura.
- [ ] Reserva memoria con `malloc` según tamaño.
- [ ] Lee archivo completo con `fread` (o hasta EOF) y verifica lectura.
- [ ] Muestra primeros N bytes en formato legible (hex).
- [ ] Libera memoria con `free()` y cierra archivo.

Ejercicio 13.5 — Lectura binaria usando arena (arenas + binario, intermedio)

Conceptos · Arenas (bump allocator), lectura binaria, bloques grandes

Consigna · Repetir el Ejercicio 13.4, pero **reservando toda la memoria necesaria desde una arena** en lugar de llamar `malloc` directamente para el buffer completo. El programa debe:

- Crear una arena con capacidad suficiente para alojar el contenido completo del archivo (o un bloque razonable si se desea leer por chunks; para este ejercicio, leer completo).
- Leer el archivo binario completo dentro de esa arena (usando el espacio de la arena).
- Mostrar tamaño y primeros N bytes (igual salida que 13.4).
- Liberar **todo junto** al final (reiniciar cursor o liberar el bloque de arena completo). **No** hacer `free` por byte/elemento.

Entrada · Ruta del archivo binario y `N`. **Salida** · Igual que Ejercicio 13.4.

Identificadores sugeridos · `arena`, `arenaBlock`, `capacity`, `offset`

Restricciones/pistas · Usar la lógica de arenas vista en Capítulo 12 (reservar bloque grande con `malloc` para la arena, avanzar cursor con alineamiento si se desea, pero alineamiento simple es suficiente para bytes). Leer el archivo completo dentro del espacio asignado por la arena (no reservar otro `malloc` grande aparte). Al finalizar, liberar **solo el bloque de la arena** (un único `free`).

Criterios de aceptación

- ☐ Crea arena con capacidad suficiente para el archivo completo.
- ☐ Lee contenido binario usando espacio de la arena (no hace `malloc` adicional para el buffer completo).
- ☐ Muestra resultados correctamente.
- ☐ Libera memoria con **un único** `free` (bloque de arena). No hay frees por partes.
- ☐ No hay fugas de memoria ni double-free.

Ejercicio 13.6 — Escritura binaria estructurada (binario, avanzado)

Conceptos · `fwrite`, estructuras, archivos binarios, persistencia

Contexto · Muchas veces queremos guardar registros en formato binario (más compacto, preserva tipos). Simplificamos: guardar un array de estructuras simples a un archivo binario, y luego leerlo para verificar.

Consigna · Definir una estructura sencilla (p.ej. `Point` con `int x, int y`) y:

1. Crear un arreglo con al menos 3 puntos.
2. Escribir ese arreglo completo a un archivo binario `puntos.bin` usando `fwrite()` (escritura de bloque: `fwrite(array, sizeof(struct), count, fp)`).
3. Leer el mismo archivo binario de vuelta a otro arreglo (reservado con `malloc`).
4. Mostrar los puntos leídos para verificar que coinciden con los originales.

Entrada · Datos de puntos (pueden estar hardcodeados para simplificar). **Salida** · Puntos originales y puntos leídos (verificación).

Ejemplo

```
Salida:  Original: (0,0), (1,2), (3,4)
         Leído:    (0,0), (1,2), (3,4)
         Verificación: OK
```

Identificadores sugeridos · `Point`, `points`, `readPoints`, `fp`, `count`

Restricciones/pistas · Usar `wb` para escribir, `rb` para leer. Escribir con un solo `fwrite` del arreglo completo (tamaño = `count * sizeof(Point)`). Leer con `fread` del mismo tamaño. Verificar cantidad leída.

Criterios de aceptación

- [ ] Escribe estructura binaria correctamente con `fwrite`.
- [ ] Lee de vuelta con `fread` y verifica coincidencia byte a byte o por campos.
- [ ] Usa modos binarios correctos (`wb`, `rb`).
- [ ] Maneja memoria con `malloc/free` adecuadamente.
- [ ] No corrompe datos al escribir/leer.

Ejercicio 13.7 — Persistencia binaria con arena (binario + arenas, avanzado)

Conceptos · Arenas, `fwrite` / `fread`, buffers desde arena

Consigna · Modificar el Ejercicio 13.6 para que **el buffer de lectura** (donde se cargan los puntos leídos) se reserve **desde una arena**, en lugar de `malloc` directo. También puede reservarse el arreglo original desde arena si se desea. El objetivo es practicar lectura binaria usando espacio de arena.

Entrada/Salida · Igual que Ejercicio 13.6. **Criterios de aceptación**

- [ ] Buffer de lectura reservado desde arena.
- [ ] Lectura con `fread` correcta.
- [ ] Verificación OK.
- [ ] Liberación con único `free` del bloque de arena (sin frees individuales).

Ejercicio 13.8 — (JEFE) Leer un archivo BMP pixel a pixel

Conceptos · Cabeceras binarias (`fread` de estructuras), little-endian, `fseek` , `ftell` , interpretación de píxeles, `malloc/free` y/o arenas (opcional), validación de formato.

Contexto · Un cliente te pide una herramienta que lea el contenido de imágenes BMP (sin compresión, 24 bits) y le diga qué colores hay en cada región de la imagen. Es un caso de "reconocimiento" simple: no hay que dibujar nada, solo **interpretar los bytes** del archivo. Como es un BMP sin compresión, cada píxel codifica su color directamente.

Consigna · Escribir un programa que:

1. Reciba la ruta de un archivo `.bmp` (idealmente generado con un editor simple o un BMP de 24 bits).
2. Lea y valide las cabeceras (**BITMAPFILEHEADER** y **BITMAPINFOHEADER**), comprobando firma `BM` , `biCompression == 0` y `biBitCount == 24` .
3. Muestre por pantalla: ancho, alto y profundidad de color.
4. Lea los píxeles completos (manejando el padding de cada fila) y muestre:
    - El color del píxel central de la imagen (en formato `R,G,B`).
    - El número de píxeles de cada uno de los colores "iguales" (si guardas un conteo).
5. Libere toda la memoria correctamente (con `free` o con una arena, según tu elección).

Entrada · Ruta del archivo `.bmp` (24 bits, sin compresión). **Salida** · Resumen legible con cabeceras y conteo de colores.

Ejemplo

```
Entrada: imagen.bmp
Salida:  Archivo: imagen.bmp
         Tamaño total: 240054 bytes (offset datos: 54)
         Ancho: 80  Alto: 100  Profundidad: 24 bpp
         Píxel central (40,50):  R=200 G=30 B=10
         Total píxeles leídos: 8000
         Colores principales:
           #1: Rojo (200,30,10)  -> 1200 px
           #2: Blanco (255,255,255) -> 6800 px
```

Identificadores sugeridos · `fileHeader`, `infoHeader`, `bfOffBits`, `biWidth`, `biHeight`, `rowSize`, `padding`, `pixelData`, `r`, `g`, `b`

Restricciones/pistas:

- Leer las cabeceras con `fread` directo a estructuras (o a buffers y ensamblar campos a mano).
- Los enteros de 32 bits son little-endian: en x86/x64 se leen directo. En otras plataformas, hay que invertir bytes.
- No olvidar el **padding** de filas a múltiplo de 4 bytes. Si no lo saltas, los colores salen corridos.
- Las filas se guardan de **abajo para arriba** (bottom-up) cuando `biHeight > 0`. Para mostrar el píxel central, podés calcular el offset directo o invertir la fila al leer.
- Podés usar una estructura simple para el conteo de colores (p.ej. guardar los 8 colores más frecuentes o un arreglo de conteos).
- Opción extra: repetir la lectura usando una **arena** para los buffers de píxeles.
- **Archivo de prueba ideal**: el BMP de 5×3 de la sección 13.8.6 (102 bytes, tres filas de color sólido). Es tan chico que verificás todo a mano: `bfOffBits = 54`, `rowSize = 16` (15 bytes de color + 1 de relleno), **3 colores × 15 píxeles** y el píxel central (2, 1) debe dar `R=34 G=177 B=76`. El archivo está en `assets/img/ejemplo-bmp-5x3.bmp`.
- La documentación del formato está en la sección 13.8. Si querés mostrar la imagen, mirá la sección 13.9 (SDL2, opcional).

Criterios de aceptación

- [ ] Valida la firma `BM` antes de continuar.
- [ ] Rechaza archivos con `biCompression != 0` o `biBitCount != 24` con mensaje claro.
- [ ] Lee y reporta correctamente ancho, alto, profundidad y offset de datos.
- [ ] Salta el padding de cada fila correctamente (imágenes de ancho no múltiplo de 4).
- [ ] Lee todos los píxeles y calcula el color del píxel central de forma correcta.
- [ ] El conteo de colores es verosímil (se puede comprobar con una imagen BMP conocida).
- [ ] Maneja archivos inexistentes o corruptos con mensajes de error claros.
- [ ] Libera memoria correctamente (con `free` o un solo `free` de la arena).
- [ ] (Extra) Implementa la versión con arena y verifica que da el mismo resultado.
- [ ] (Extra) Muestra la imagen en una ventana con SDL2 (ver sección 13.9).