Capítulo 13 — Entrada y salida estándar (sin archivos)

> *Capítulo intermedio, pero **escrito para que se entienda de verdad**. Se apoya en variables (`02`), operadores (`03`), `if` y bucles (`04`), funciones (`05`) y arreglos y cadenas (`06`).*
>
> *Es el **puente** hacia el capítulo de archivos (`14-archivos`). Todo lo que aprendas acá sobre `stdin` , búferes y `EOF` lo vas a reusar tal cual para leer archivos. La única diferencia será que, en vez de leer del teclado, vas a leer de un archivo.*

Cómo leer este capítulo

Bienvenido al capítulo que separa a los que "hacen que ande" de los que **entienden qué pasa**. Acá no vas a memorizar tablas: vas a **mirar**, **probar** y **romper** cosas hasta que los conceptos te queden pegados.

Está dividido en **ocho partes**. Después de cada idea importante hay una **práctica** —uno o varios ejercicios cortos— para que la uses ya mismo, antes de avanzar. No saltees las prácticas: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	Los tres flujos	Tu programa tiene tres tuberías: una de entrada y dos de salida.
2	Leer byte a byte	`getchar` saca de a uno, y `EOF` avisa cuándo se acabó.
3	Leer líneas	`fgets` saca una línea entera sin desbordarse.
4	El búfer	Por qué un `Enter` cambia todo, y por qué `scanf` deja basura.
5	Leer con formato	`scanf` y su mini-lenguaje de formatos.
6	Caracteres	Por qué una `á` pesa dos bytes.
7	Robustez	Que tu programa no se rompa nunca.
8	Proyecto final	Construís tu propia consola de comandos.

Los **ejercicios** se numeran en orden: `13.1` , `13.2` , `13.3` , y así. Cada uno trae su **escenario de prueba** (qué entra y qué tiene que salir) para que puedas **comprobar** tu solución.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada cosa, pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

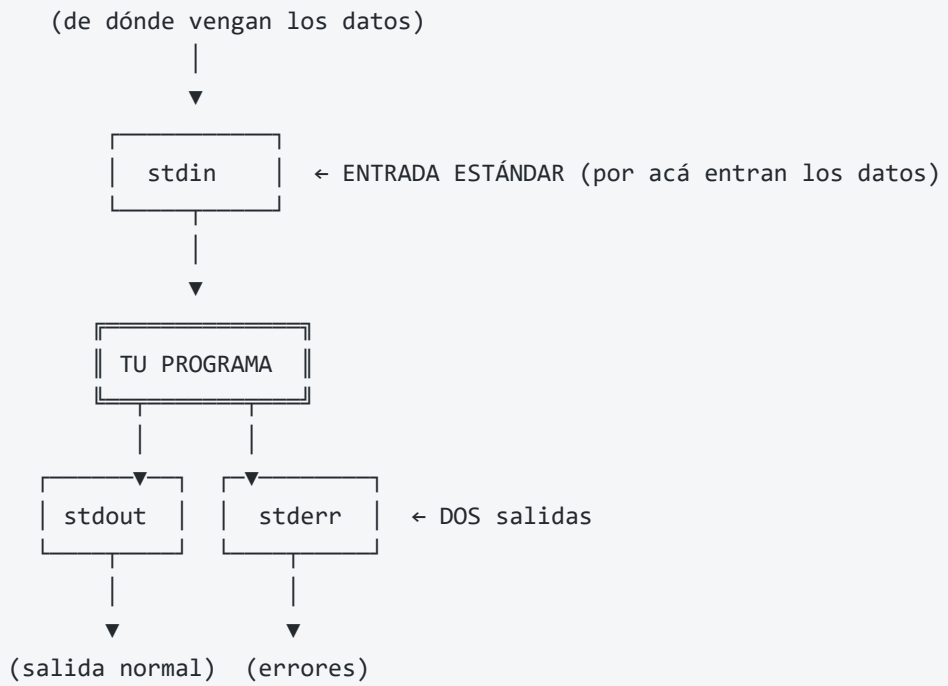
PARTE 1 — Tu programa tiene tres tuberías

Antes de tocar una sola función, entendamos el terreno. Es más simple de lo que parece.

1.1 El día que descubrí que mi programa no sabe de dónde lee

Imaginate que tu programa es una **fábrica**. La fábrica no sabe si la materia prima llega en camión, en tren o por un caño: solo sabe que **algo entra** por una entrada, se procesa, y **algo sale** por las salidas. En C, esa entrada y esas salidas son **tuberías** (en inglés, *streams*, "flujos").

Todo programa en C nace con **tres tuberías ya conectadas**:



- `stdin` es la tubería de **entrada**. Por ahí entran los datos.
- `stdout` es la tubería de **salida normal**. Por ahí sale el resultado del programa.
- `stderr` es la tubería de **errores**. Por ahí salen los mensajes de problema.

Esas tres tienen un nombre con `st` adelante porque viene de *standard*: **estándar**. No las abrís vos. Existen desde que arranca `main`, como la luz de tu casa: no la instalás, ya está.

1.2 `FILE *`: la ficha de la tubería

Cada tubería, en el código, se maneja con algo que se llama `FILE *`. Podés pensarlo como la **factura** o la **ficha** de esa tubería: un papelito que dice "esta es la tubería de entrada, así está configurada, así se va leyendo". No es la tubería en sí: es su **identificación**.

```c
#include <stdio.h>

int main(void) {
    /* stdin, stdout y stderr ya vienen declarados en <stdio.h>.
       Son de tipo FILE *. No hay que abrirlos ni cerrarlos. */
    (void)stdout;
    return 0;
}
```

> ***¿Por qué `FILE *` y no simplemente "el teclado"?*** *Porque tu programa* ***no habla con el teclado****: habla con la* ***tubería****. Eso es lo que lo hace genial. La misma `stdin` puede estar conectada al teclado, a un archivo o a otro programa, y tu código* ***no cambia ni una línea****. Eso lo vas a ver ya mismo, en la parte más divertida.*

1.3 El truco más lindo de la consola: la misma tubería, conectada a cualquier cosa

Acá está la magia. Vos escribís **un** programa, y el **shell** (la terminal, la ventanita donde escribís comandos) decide **a qué** conecta sus tuberías. Mirá cómo cambia todo sin tocar el programa.

Vamos a usar `cat`, un programa clásico que simplemente copia lo que entra a lo que sale. Es como un **espejo**: lo que le des, te lo devuelve.

Forma 1: entrada desde el teclado. El shell conecta la `stdin` de `cat` a tu teclado.

```
$ cat
hola mundo          ← lo que escribís vos
hola mundo          ← lo que devuelve cat (el eco)
esto es una prueba
esto es una prueba
(Ctrl+Z y Enter en Windows / Ctrl+D en Linux y macOS para terminar)
```

Forma 2: entrada desde una tubería. Con el símbolo `|` (la "tubería"), conectás la **salida** de un programa a la **entrada** de otro. Fijate que `echo` **escribe** y `cat` **lee**, y ninguno de los dos sabe del otro.

```
$ echo "hola mundo" | cat
hola mundo
```

Forma 3: entrada desde un archivo. Con el símbolo `<`, conectás la `stdin` de `cat` a un archivo.

```
$ cat < datos.txt
primera línea del archivo
segunda línea del archivo
```

> ***Momento "ajá".*** *En los tres casos,* `cat` *hizo lo mismo internamente: leer de* `stdin` *y escribir en* `stdout`*.* ***Nunca supo*** *si leía del teclado, de otro programa o de un archivo. El shell cambió* ***a qué estaba conectada la tubería****, nada más.*

1.4 Y las salidas también se pueden desviar

Igual que la entrada, podés desviar las **salidas**. El símbolo `>` manda la salida (normal) a un archivo en vez de la pantalla:

```
$ cat < datos.txt > copia.txt      # leo de datos.txt y escribo en copia.txt
$ cat copia.txt
primera línea del archivo
segunda línea del archivo
```

Ahora, la parte que casi nadie sabe al principio: `>` **desvía** `stdout`**, pero NO** `stderr`. Los errores **siguen saliendo por su propio canal**. Probémoslo pidiendo un archivo que no existe:

```
$ ls archivo_que_no_existe
ls: cannot access 'archivo_que_no_existe': No such file or directory

$ ls archivo_que_no_existe > salida.txt
ls: cannot access 'archivo_que_no_existe': No such file or directory   ← ¡sigue en
pantalla!
$ cat salida.txt
                                                    ← salida.txt está VACÍO: eso fue
un error
```

El mensaje de error **no** fue a `salida.txt`, porque pertenece a `stderr`. Eso es **exactamente** para lo que existe `stderr`: así podés guardar la salida útil en un archivo **sin perder** los errores de vista.

Símbolo	Qué desvía	Ejemplo
`<`	La **entrada** (`stdin`) desde un archivo	`prog < datos.txt`
`>`	La **salida normal** (`stdout`) a un archivo	`prog > out.txt`
`2>`	La **salida de error** (`stderr`) a un archivo	`prog 2> err.txt`
`|`	La **salida** de uno a la **entrada** de otro	`echo hola | prog`
`2>&1`	Manda `stderr` **al mismo lugar** que `stdout`	`prog > todo.txt 2>&1`

> ***Truco de laboratorio.*** *Para ver la diferencia en vivo, corré cualquier comando tres veces:* `cmd`*,* `cmd > out.txt 2> err.txt`*, y* `cmd > todo.txt 2>&1`*. Después mirá qué cayó en cada archivo. Es la mejor forma de* ***sentir*** *los canales separados en la mano.*

1.5 Los tres flujos vistos desde el código

Desde C, los tres son `FILE *`, y los tres se pueden usar tanto para leer como para escribir (aunque en la práctica `stdin` se lee y las otras dos se escriben). `printf` escribe en `stdout`; `fprintf(stderr, ...)` escribe en `stderr`:

```c
#include <stdio.h>

int main(void) {
    /* fprintf: escribe con formato en un flujo cualquiera.
       Primer argumento: el flujo (acá stdout, que es igual que usar printf).
       El resto: el formato y los datos, exactamente como printf. */
    fprintf(stdout, "Esto es la salida normal\n");

    /* Mismo fprintf, pero apuntando a stderr: esto es un mensaje de error. */
    fprintf(stderr, "Esto es un mensaje de error\n");

    return 0;
}
```

Si lo corrés y desviás solo la salida normal, el mensaje de error **queda en pantalla**:

```
$ ./hola > out.txt
Esto es un mensaje de error          ← quedó en pantalla (stderr)
$ cat out.txt
Esto es la salida normal             ← esto sí fue al archivo
```

1.6 Un detalle que sorprende: el orden no siempre es el que escribiste

`stderr` normalmente **no espera**: envía cada mensaje al instante. `stdout`, en cambio, suele **acumular** (guardar en un búfer) y enviar todo junto, sobre todo cuando la salida va a un archivo o a un caño. Resultado: si mezclás escrituras a `stdout` y a `stderr`, el orden en pantalla puede no

ser el orden en que las programaste. Es un motivo más para **no mezclar** la salida normal con los errores.

1.7 "Estándar" y "multiplataforma": dos palabras que valen oro

Cuando decimos que `getchar` o `fgets` son **estándar**, queremos decir que están **descritas en el contrato del lenguaje C** (el estándar ISO C). Ese contrato define el **nombre**, los **parámetros** y **qué hacen**, y **cualquier** implementación que cumpla el estándar (GCC, Clang, MSVC…) las ofrece con el mismo significado. Por eso el **mismo código** corre en Windows, Linux y macOS: eso es ser **multiplataforma**.

Pero atención: el estándar garantiza el **contrato**, no los **detalles** de cada máquina. Mirá la diferencia:

Garantizado por el estándar (vale en todos lados)	Depende del entorno (cambia según la máquina)
Los nombres y parámetros de las funciones de `<stdio.h>`	Qué tecla produce `EOF`: Ctrl+Z (Windows) o Ctrl+D (Linux/macOS)
Cuántas asignaciones hizo `scanf`, qué devuelve `fgets`…	Si el teclado muestra o no lo que escribís (el "eco")
Que `EOF` es **negativo**, distinto de todo byte (0–255)	La **codificación** de los caracteres (UTF-8, Latin-1, CP1252…)
Que las cadenas terminan en `'\0'`	El tamaño de los búferes y el momento en que se vuelcan

> ***Para guardar.*** *Tu programa lee y escribe* ***tuberías****. El* ***shell*** *decide a qué está conectada cada una. Tu código escribe* ***una vez*** *y funciona en teclado, caño y archivo… siempre que respetes el* ***contrato*** *de cada función. Lo que cambia de máquina a máquina son los detalles, no el contrato.*

Práctica de la Parte 1

Ejercicio 13.1 — Separar la salida útil de los avisos

Conceptos · `stdout`, `stderr`, `fprintf`, redirección de salidas

Contexto · Las herramientas de línea de comandos separan la salida **útil** (`stdout`) de los **diagnósticos** (`stderr`), para que puedas guardar una sin perder los otros. Simplificación: un programa que "cierra un trabajo" y tiene que avisar de un problema sin ensuciar el resultado.

Consigna · Escribí un programa que simule el cierre de un reporte: manda las líneas informativas (dos o tres, por ejemplo "Iniciando reporte...", "Procesando 128 registros...", "Reporte generado correctamente.") a `stdout`, y el aviso de problema (por ejemplo "Aviso: 3 registros tenían fecha inválida.") a `stderr`. Nunca al revés.

Entrada · Ninguna: el programa no lee nada. **Salida** · Las líneas informativas por `stdout`; el aviso por `stderr`.

Escenario de prueba

```
$ ./reporte > salida.txt 2> errores.txt
$ cat salida.txt
Iniciando reporte...
Procesando 128 registros...
Reporte generado correctamente.

$ cat errores.txt
Aviso: 3 registros tenían fecha inválida.
```

Criterios de aceptación

- [ ] Las líneas informativas salen **solo** por `stdout`.
- [ ] El aviso de problema sale **solo** por `stderr`.
- [ ] Al correr con `>` y `2>`, cada mensaje cae en su archivo y ninguno aparece en el otro.
- [ ] Usa `fprintf` indicando el flujo (`stdout` / `stderr`) en cada escritura.

Ejercicio 13.2 — Detective de flujos

Conceptos · `stdout`, `stderr`, redirección (`>`, `2>`, `2>&1`), tuberías

Contexto · Antes de creer que entendiste los flujos, hay que **verlos**. Vas a conectar el mismo programa de varias maneras y comparar qué termina dónde.

Consigna · Tomá el programa de 13.1 (o escribí uno mínimo que mande una línea a `stdout` y otra a `stderr`) y corrélo de estas formas, anotando para cada una **qué queda en pantalla** y **qué en cada archivo**:

1. sin redirección;
2. `./programa > out.txt`;
3. `./programa 2> err.txt`;
4. `./programa > out.txt 2> err.txt`;
5. `./programa > todo.txt 2>&1`.

Al final, respondé: ¿cambió el **programa**? ¿cambió el **resultado**? ¿quién decidió a dónde va cada línea?

Entrada · Ninguna: el programa no lee nada. **Salida** · Dos líneas (una por `stdout`, una por `stderr`), que caen en distintos destinos según cómo las redirijas.

Escenario de prueba

```
$ ./dos > out.txt 2> err.txt
$ cat out.txt
salida normal

$ cat err.txt
mensaje de error
```

```
$ ./dos > todo.txt 2>&1
$ cat todo.txt
salida normal
mensaje de error
```

Criterios de aceptación

- ☐ El mismo binario se comporta distinto **solo** por cómo el shell conecta sus salidas (sin recompilar).
- ☐ Con `> out.txt 2> err.txt`, cada mensaje cae en su archivo.
- ☐ Con `2>&1`, los dos mensajes terminan en el mismo archivo.
- ☐ Sabés explicar qué cambió (la conexión) y qué **no** cambió (el programa).

PARTE 2 — Leer byte a byte: `getchar` y el famoso `EOF`

Ya sabemos de dónde vienen los datos. Ahora empecemos a **sacarlos**, de a poquito.

2.1 `getchar`: el caramelito de a uno

`getchar()` hace una sola cosa: **saca un byte** de `stdin`. Nada más. Cada vez que la llamás, te da el próximo byte de la entrada. Como sacar caramelos de un frasco, **de a uno por vez**.

```c
#include <stdio.h>

int main(void) {
    /* getchar: lee UN byte de stdin y lo devuelve.
       Lo guardamos en un int (¡no en un char!, ver 2.4). */
    int c = getchar();
    (void)c;
    return 0;
}
```

Si tecleás `hola` y apretás `Enter`, no hay **un** byte esperando: hay **cinco** (`h`, `o`, `l`, `a` y el `'\n'` del Enter). `getchar` te dará uno por llamada:

```
1ª llamada a getchar()  →  'h'
2ª llamada a getchar()  →  'o'
3ª llamada a getchar()  →  'l'
4ª llamada a getchar()  →  'a'
5ª llamada a getchar()  →  '\n'   (el Enter que apretaste)
6ª llamada a getchar()  →  EOF    (ya no hay más)
```

Lo mismo pasa con los **números**, y ahí está la trampa más grande de todas: para `getchar`, un número **no existe**. Solo existen sus **dígitos**, uno por uno. Si tecleás `304` y apretás `Enter`, no hay un "trescientos cuatro" esperando: hay **cuatro** bytes (`'3'`, `'0'`, `'4'` y el `'\n'`).

```
Tecleás: 304 <Enter>

1ª llamada a getchar()  →  '3'    (el byte del dígito tres)
2ª llamada a getchar()  →  '0'    (el byte del dígito cero)
3ª llamada a getchar()  →  '4'    (el byte del dígito cuatro)
4ª llamada a getchar()  →  '\n'   (el Enter que apretaste)
5ª llamada a getchar()  →  EOF    (ya no hay más)
```

Ojo con esto. *`getchar` te devuelve el **byte del carácter**, no el valor del número. Cuando lee `'3'`, te da un byte (cuyo valor es `51`), no el número `3`. Armar el número `304` a partir de esos dígitos es un trabajo **tuyo**, no de `getchar`. Por eso, para leer números "de verdad", más adelante vas a usar `scanf` (Parte 5).*

2.2 El patrón que vas a repetir mil veces

Casi todos los programas que leen byte a byte usan el mismo esqueleto: leer y **procesar**, en un bucle, hasta que se acabe la entrada.

```c
#include <stdio.h>

int main(void) {
    int c;

    /* Bucle clásico: leo un byte; si no es EOF, lo proceso y repito.
       getchar() devuelve EOF cuando ya no queda nada por leer. */
    while ((c = getchar()) != EOF) {
        putchar(c);   /* putchar: escribe UN byte en stdout (el espejo de getchar) */
    }

    return 0;
}
```

Este programita es un espejo: copia la entrada a la salida. Es, básicamente, nuestro `cat`.

2.3 `EOF` : el timbre que dice "se acabó la fila"

`EOF` significa *End Of File* (fin de archivo), pero no lo interpretes como "fin de un archivo de disco". Es más general: significa **"no hay más datos para leer"**, venga de donde venga.

- Si leés de un **archivo** y llegaste al final → `EOF` .
- Si leés de una **tubería** y el otro programa terminó → `EOF` .
- Si leés del **teclado** y apretás **Ctrl+Z + Enter** (Windows) o **Ctrl+D** (Linux/macOS) → `EOF` .

Pensalo como el **timbre del recreo**: no es un alumno más en la fila, es la **señal** de que la fila se terminó. `getchar` devuelve ese timbre como un valor **especial**: `EOF` .

```
Byte 'h'  → valor 104      (es un dato)
Byte '\n' → valor 10       (es un dato)
EOF       → valor -1       (¡no es un dato! es "no hay más")
```

2.4 La trampa N.º 1: guardar `getchar` en un `char`

Fijate en algo raro: `getchar` **no** devuelve un `char` , devuelve un `int` . ¿Por qué?

Porque tiene que poder representar **256 bytes posibles** (de `0` a `255`) **y además** un valor extra: `EOF` (`-1`). Un `char` no tiene lugar para ese "valor extra" sin pisar un byte real. Si guardás el resultado en un `char` , pasan cosas feas:

- El byte `0xFF` (que en algunos teclados/enviros es la letra `ÿ`) puede convertirse en `-1` , ¡y `-1` es `EOF` ! Entonces un **dato real** se confunde con "se acabó".
- O al revés: comparás contra `EOF` y la comparación **nunca** da verdadera, aunque la entrada terminó.

La regla, entonces: el resultado de `getchar` (y de `getc` / `fgetc`) se guarda **siempre en un** `int`, y contra `EOF` se compara **en ese** `int`.

```c
int c;                      /* int, NO char */
while ((c = getchar()) != EOF) {
    /* acá c es un byte válido entre 0 y 255 */
}
```

2.5 `getchar`, `getc` y `fgetc`: tres nombres, casi lo mismo

Existen tres primos que leen **un byte**, y hacen lo mismo:

```c
int getchar(void);              /* lee SIEMPRE de stdin */
int getc(FILE *stream);         /* lee de CUALQUIER flujo (ej.: getc(stdin)) */
int fgetc(FILE *stream);        /* igual que getc, pero siempre es función */
```

- `getchar()` es la versión fija a `stdin`: comodísima para la consola.
- `getc(stdin)` y `fgetc(stdin)` hacen lo mismo que `getchar()`, pero sirven para **cualquier** flujo (muy útil cuando llegues a los archivos).
- La única diferencia fina: `getc` puede estar implementada como **macro** y `fgetc` **siempre** es función. En la práctica, usá `getchar` para la consola y `fgetc(fp)` para un archivo.

Práctica de la Parte 2

Ejercicio 13.3 — Contador de caracteres seguro

Conceptos · `getchar`, `int` vs `char`, `EOF`, bytes altos (`0xFF`)

Contexto · Contar bytes de la entrada es lo que hace la herramienta `wc -c`. Simplificación: contar **bytes** leídos de `stdin` hasta `EOF`, sin confundir un byte `0xFF` con el fin de archivo.

Consigna · Leer de `stdin` **byte a byte** con `getchar()` hasta `EOF` y contar cuántos bytes se leyeron. La variable de lectura **y** el contador deben ser `int`.

Entrada · Cualquier secuencia de bytes (texto o no) hasta `EOF`. **Salida** · Una línea con la cantidad de bytes leídos.

Escenario de prueba

```
printf 'hola' | ./contador
Caracteres leídos: 4
```

```
printf '\xff\xffh' | ./contador        (dos bytes 0xFF y una 'h')
Caracteres leídos: 3
```

> ***Para comprobar que lo hiciste bien:*** *el conteo debe dar* ***3*** *aunque haya bytes* `0xFF`*. Si guardaste en un* `char`*, el programa corta en el primer* `0xFF` *y te da* ***0****.*

Criterios de aceptación

- [ ] Usa `int` para la lectura y para el contador.
- [ ] Cuenta **todos** los bytes, incluidos `0x00` y `0xFF`.
- [ ] Termina limpio en `EOF`, sin bucle infinito.
- [ ] El resultado coincide con el tamaño real de la entrada.

Ejercicio 13.4 — ¿Cuántos saltos de línea hay?

Conceptos · `getchar`, comparación de bytes, `'\n'`, `EOF`

Contexto · Los editores cuentan líneas contando fines de línea. Simplificación: contar cuántos bytes `'\n'` hay en la entrada (que es, ni más ni menos, cuántas líneas terminadas tiene).

Consigna · Leer de `stdin` byte a byte y contar **solo** los bytes que sean `'\n'`. Al terminar, mostrar el total.

Entrada · Texto hasta `EOF`. **Salida** · La cantidad de saltos de línea.

Escenario de prueba

```
printf 'uno\ndos\ntres\n' | ./lineas
Saltos de línea: 3
```

```
printf 'sin salto al final' | ./lineas
Saltos de línea: 0
```

Criterios de aceptación

- ☐ Lee con `getchar` en un `int` y corta ante `EOF`.
- ☐ Cuenta **solo** `'\n'` (no otros bytes).
- ☐ Los dos escenarios dan el resultado esperado.
- ☐ No hay bucle infinito si la entrada no termina en `'\n'`.

PARTE 3 — Leer líneas enteras: `fgets`

Leer de a un byte está bien, pero muchas veces querés "una línea entera de una". Para eso existe `fgets`.

3.1 El cucharón que llena el balde

`fgets` es como un **cucharón**: saca de la tubería hasta **llenar un balde** (un arreglo de `char`) o hasta encontrar el fin de la línea, lo que pase primero.

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

- `s`: el **balde** (tu arreglo de `char`) donde se guarda lo que leyó.
- `n`: el **tamaño** del balde.
- `stream`: de qué tubería leer (`stdin`, en la consola).
- **Devuelve**: `s` si leyó algo, o `NULL` si llegó al final o hubo error.

3.2 Tres reglas de oro de `fgets`

1. **Lee como máximo `n - 1` caracteres.** Reserva un lugar para el terminador `'\0'`. Por eso un balde de 20 guarda **19** letras + el `'\0'`.
2. **Se detiene en `'\n'` … y lo guarda.** A diferencia de lo que haría un humano, `fgets` **deja el `'\n'` adentro** del balde. Es una fuente clásica de sorpresas (ver Parte 4).
3. **Nunca se desborda si le pasás el tamaño real.** Como sabe cuánto mide el balde, para cuando se llena. Es la función **segura** para leer por consola.

3.3 Leer líneas en un bucle

```c
#include <stdio.h>

int main(void) {
    char line[100];

    /* fgets devuelve NULL cuando ya no hay más líneas (EOF) o hay error.
       Le paso sizeof line, así nunca escribe fuera del arreglo. */
    while (fgets(line, sizeof line, stdin) != NULL) {
        printf("Leí: %s", line);   /* line ya incluye el '\n', por eso no lo agrego */
    }

    return 0;
}
```

3.4 Líneas más largas que el balde

Si la línea es más larga que el balde, `fgets` **la corta**: llena el balde hasta `n-1` y sigue en la próxima llamada. No se rompe nada, pero tu programa ve la línea **partida**. Para darte cuenta, el truco es: si el balde se llenó y **NO** terminó en `'\n'`, la línea era más larga de lo que entra.

```c
char line[20];
fgets(line, sizeof line, stdin);
int len = strlen(line);                 /* strlen necesita <string.h> */
if (len > 0 && line[len - 1] != '\n') {
    /* el balde se llenó sin encontrar el '\n': la línea venía cortada */
}
```

3.5 Cuando la línea no entra: descartar el resto

Detectar que una línea se cortó es solo la mitad. La otra mitad es **descartar lo que sobró**: si no, esa parte se va a leer como si fuera una línea nueva (y vas a procesar datos fantasma).

Después de detectar el corte, "comete" lo que queda de la línea hasta el `'\n'`:

```c
char line[20];
fgets(line, sizeof line, stdin);

int len = strlen(line);
if (len > 0 && line[len - 1] != '\n') {
    /* La línea era más larga que el balde: tiro lo que sobra
       hasta consumir el '\n' (o llegar a EOF). */
    int c;
    while ((c = getchar()) != '\n' && c != EOF) {
        /* vacío a propósito: solo descarto */
    }
}
```

Ese bucle es un **patrón que vas a reusar**: en la Parte 4 vuelve a aparecer, con la misma forma, para limpiar los residuos que deja `scanf`. Guardátelo.

Práctica de la Parte 3

Ejercicio 13.5 — Protección contra desbordamiento

Conceptos · `fgets` con búfer fijo, truncamiento, detección de líneas largas, descarte del resto

Contexto · Un lector de comandos tiene que aceptar entradas más largas que su búfer **sin** desbordar la memoria. Simplificación: búfer **estático de 20 bytes** y aviso de recorte.

Consigna · Leer un comando con `fgets` en un búfer estático de **20 bytes**. Si la línea era más larga que el búfer, avisar que se recortó y **descartar el resto** con el patrón de 3.5, para que no se lea como una línea nueva.

Entrada · Una línea (posiblemente más larga que el búfer). **Salida** · El comando leído y, si corresponde, el aviso de recorte.

Escenario de prueba

```
Entrada:  una orden extremadamente larga
Salida:
Comando: "una orden extremada"
Aviso: la línea era más larga que el búfer (20 bytes) y fue recortada.
```

> *El búfer de 20 guarda **19** caracteres + `'\0'` : por eso se ve `una orden extremada` (19 letras). El resto (`mente larga`) se descarta.*

Criterios de aceptación

- [ ] `fgets` recibe el tamaño real del búfer (`sizeof`).
- [ ] Entradas cortas se leen completas.
- [ ] Entradas largas se recortan a 19 caracteres **y** se avisa.
- [ ] El resto de la línea se descarta; no contamina la lectura siguiente.
- [ ] No hay desbordamiento ni comportamiento indefinido.

Ejercicio 13.6 — Contador de palabras

Conceptos · `fgets` , recorrido de una cadena, espacios, bucles

Contexto · Contar palabras es el trabajo de `wc -w` . Simplificación: contar cuántas "palabras" (sucesiones de caracteres que no son espacio) hay en la entrada, leyendo línea por línea.

Consigna · Leer la entrada con `fgets` y contar las palabras. Una palabra empieza cuando pasás de "espacio" a "no espacio". Contá también la última palabra si la línea no termina en espacio.

Entrada · Texto hasta `EOF` . **Salida** · La cantidad de palabras.

Escenario de prueba

```
printf 'hola mundo\nesto es una prueba\n' | ./palabras
Palabras: 6
```

```
printf '   muchos    espacios   \n' | ./palabras
Palabras: 2
```

Criterios de aceptación

- [ ] Lee con `fgets` y comprueba que no devolvió `NULL`.
- [ ] Cuenta las palabras correctamente, sin importar cuántos espacios haya de más.
- [ ] Los dos escenarios dan el resultado esperado.
- [ ] No asume que toda línea termina en espacio.

PARTE 4 — El búfer: el villano de la película

Esta es **la parte más importante** del capítulo. Si entendés el búfer, se te acaban los misterios.

4.1 ¿Por qué `getchar` parece quedarse dormido?

Escribí este programa y corrélo. Va a leer un byte y mostrarlo:

```c
#include <stdio.h>

int main(void) {
    printf("Escribí algo: ");
    int c = getchar();
    printf("Leí el byte: '%c'\n", c);
    return 0;
}
```

Si lo corrés y escribís `hola`, **no pasa nada** hasta que apretás `Enter`. Recién ahí el programa continúa. ¿Por qué? Porque hay **dos búferes** en el medio, y entenderlos es entender todo.

4.2 Los dos búferes (y por qué existen)

Un **búfer** es una **zona de memoria temporal** que junta datos antes de pasarlos, para no estar yendo y viniendo al dispositivo en cada byte (que sería lentísimo). Hay **dos** búferes en serie:

```
tus teclas → [ búfer de la TERMINAL ] → [ búfer de la LIBRERÍA C ] → tu getchar()
              (lo maneja el sistema)     (lo maneja tu programa)
              se vacía con ENTER         entrega byte a byte
```

- **Búfer de la terminal:** lo maneja el **sistema operativo**. Acumula lo que tecleás y **solo lo entrega cuando apretás** `Enter`. Por eso `getchar` "espera".
- **Búfer de la librería C:** lo maneja **tu programa**. Recibe lo que entregó el sistema y lo entrega a tus `getchar`, **de a un byte**.

4.3 Modo "canónico": el teclado entrega por líneas

Ese comportamiento —"entregá todo junto recién con `Enter`"— se llama **modo canónico** y es el **modo por defecto** de las terminales. ¿Se puede cambiar para que cada tecla se entregue al instante? Sí, se llama **modo no canónico** (o *raw*), y lo usan los juegos de texto y los editores de pantalla completa.

> ***Pero ojo:*** *el modo* ***no lo elige*** `getchar`. *Lo elige la* ***terminal****, y se configura con APIs del sistema (termios en Linux,* `SetConsoleMode` *en Windows) que* ***no son estándar de C****. Por eso "leer una tecla sin apretar* `Enter` *"* ***no es portable****. Con las funciones de este capítulo, siempre vas a necesitar el* `Enter`.

4.4 El bug más famoso del principiante: `scanf` deja el `\n` pendiente

Ahora sí, el villano. Mirá este clásico:

```c
int age;
char name[50];

scanf("%d", &age);                      /* el usuario escribe: 30 <Enter> */
fgets(name, sizeof name, stdin);        /* ¡lee una línea vacía!          */
```

¿Qué pasó? `scanf("%d", ...)` lee los dígitos `3` y `0`, y **para** en el primer carácter que no es dígito: el `'\n'` del Enter. Ese `\n` **queda en el búfer**. Después, `fgets` ve que "hay una línea pendiente" (que solo contiene ese `\n`) y la lee: `name` queda **vacío**. El nombre nunca se leyó.

> ***No es un bug de*** `scanf`***:*** *es su* ***contrato****.* `%d` *lee* ***el número****, no "la línea". Lo que sobra queda ahí para la próxima lectura. Y la próxima lectura sos vos.*

4.5 La solución: "comerse" el resto de la línea

Antes de leer una línea con `fgets`, hay que **descartar** lo que quedó pendiente. El patrón es:

```c
/* Descarta lo que quede en la línea actual, hasta consumir el '\n'. */
int c;
while ((c = getchar()) != '\n' && c != EOF) {
    /* vacío a propósito: solo tiramos los caracteres */
}
```

El bucle lee y **descarta** hasta encontrar el `'\n'` (lo consume) o hasta `EOF`. Después de esto, el búfer queda limpio y "alineado" al comienzo de la línea siguiente.

4.6 Por qué `fflush(stdin)` NO se usa (aunque te lo recomienden)

Mucha gente "limpia" el búfer con `fflush(stdin)`. **No lo hagas.** `fflush` está definido por el estándar **solo para flujos de salida** (para vaciar lo que escribiste). Aplicado a la **entrada**, el comportamiento es **indefinido**:

```c
fflush(stdin);   /* ¡NO PORTABLE! El estándar no define qué hace */
```

Algunos compiladores (MSVC, en Windows) le dan un sentido útil, y por eso el código con `fflush(stdin)` "funciona" en esa máquina. Pero **no es estándar** y puede fallar en otra. La forma **portable** de limpiar es el bucle de 4.5.

Práctica de la Parte 4

Ejercicio 13.7 — Corrección de búfer tras `scanf`

Conceptos · `scanf`, `fgets`, `\n` pendiente, limpieza del búfer

Contexto · El error más común al mezclar `scanf` y `fgets`: el `\n` que `scanf` deja pendiente "come" la línea siguiente. Simplificación: leer un número y **después** un nombre, sin perderlo.

Consigna · Pedir un número entero con `scanf` y luego una cadena corta con `fgets`. Implantá la **limpieza del búfer** intermedia (la de 4.5) para que el `\n` pendiente no arruine el nombre.

Entrada · Dos líneas: un número y un texto. **Salida** · El número y el nombre, cada uno en su línea.

Escenario de prueba

```
Entrada:
42
Ana
Salida:
Número: 42
Nombre: Ana
```

> ***Para comprobar que lo hiciste bien:*** *sin la limpieza, la salida sería* `Nombre:` *(vacío), porque* `fgets` *leería el* `'\n'` *pendiente en vez de* `Ana`.

Criterios de aceptación

- [ ] Comprueba el retorno de `scanf`.
- [ ] Limpia el búfer pendiente de forma **portable** (bucle de lectura), **no** con `fflush`.
- [ ] `fgets` obtiene el nombre, **no** una línea vacía.
- [ ] Muestra número y nombre correctos.

Ejercicio 13.8 — Cuestionario de datos mixtos

Conceptos · `scanf`, `getchar`, `fgets`, saltear blancos, orden de lectura, residuos

Contexto · Un alta de usuario mezcla un ID numérico, una marca sí/no y un nombre completo (con espacios). Simplificación: capturar los tres desde la misma entrada, sin residuos entre lecturas.

Consigna · Leer, en este orden y desde la misma entrada:

1. el **ID** numérico con `scanf("%d", ...)`;
2. **saltar los blancos** que quedaron después del número (espacios y tabs), leyendo con `getchar()`;
3. la **marca** `s` / `n` (un solo carácter) con `getchar()`;
4. **descartar el resto de la línea** con el patrón de 4.5, para no arrastrar residuos;
5. el **nombre completo** con `fgets()` (puede tener espacios).

Garantizá que **no** queden caracteres residuales entre una lectura y la siguiente.

Entrada · Primera línea: `ID marca` (separados por un espacio); segunda línea: el nombre.
Salida · Los tres datos, etiquetados.

Escenario de prueba

```
Entrada:
1001 s
Juan Gómez
Salida:
ID: 1001
Activo: s
Nombre: Juan Gómez
```

Criterios de aceptación

- [ ] Lee los tres datos en el orden pedido, desde la misma entrada.
- [ ] El nombre conserva los espacios internos (`Juan Gómez`).
- [ ] No quedan residuos: la lectura del nombre **no** está "corrida".
- [ ] Comprueba los valores de retorno.

Ejercicio 13.9 — Menú de opciones de longitud fija

Conceptos · `getchar`, consumir el resto de la línea, bucles, validación

Contexto · Los menús de consola leen **una** opción. Si el usuario escribe una cadena larga, hay que quedarse con el primer carácter y **no** arrastrar el resto a la próxima vuelta.

Consigna · Mostrar un menú (`[A] alta [B] baja [C] consulta [Q] salir`), leer la opción con `getchar()` y, si el usuario escribió **más de un carácter**, **consumir el resto de la línea** antes de la iteración siguiente. Repetir hasta `Q` o hasta `EOF`.

Entrada · Líneas de opciones. **Salida** · El eco de cada opción elegida y, al final, el cierre.

Escenario de prueba

```
Entrada:
xyz
C
(Ctrl+Z / Ctrl+D para terminar)

Salida:
[A] alta  [B] baja  [C] consulta  [Q] salir
> Elegiste: x
[A] alta  [B] baja  [C] consulta  [Q] salir
> Elegiste: C
[A] alta  [B] baja  [C] consulta  [Q] salir
> Fin del menú
```

> *Fijate que la entrada `xyz` elige `x` y **descarta** `yz` : por eso la vuelta siguiente arranca limpia.*

Criterios de aceptación

- [ ] Lee un **único** carácter por iteración.
- [ ] Si se escribió más de un carácter, consume el resto de la línea.
- [ ] La opción siguiente **no** está contaminada por residuos.
- [ ] Termina con `Q` y también ante `EOF` , sin bucle infinito.

PARTE 5 — Leer con formato: `scanf` y sus mañas

`fgets` te da la línea. `scanf` te da **el dato que le pedís**, en el formato que le indiques. Es potente y cómodo… y traicionero si no respetás sus reglas.

5.1 La idea: un mini-lenguaje de formatos

`scanf` recibe una **cadena de formato** donde le describís **qué** querés leer. Cada conversión empieza con `%`:

```c
int n;
double price;
char letter;

scanf("%d", &n);          /* lee un entero decimal              */
scanf("%lf", &price);     /* lee un número con coma (double)    */
scanf(" %c", &letter);    /* lee UN carácter (el espacio salta blancos) */
```

Siempre se le pasa la **dirección** de la variable (con `&`): `&n`, no `n`. Suena obvio, pero es el error N.º 1.

5.2 Qué devuelve `scanf`: el dato clave para no romperte

`scanf` **no** devuelve lo que leyó. Devuelve **cuántas variables llenó**:

- `scanf("%d", &n)` → devuelve `1` si leyó un número bien.
- `scanf("%d %d", &a, &b)` → devuelve `2` si leyó los dos.
- `0` si el primer dato **no coincidía** con el formato (por ejemplo, pedís `%d` y hay `abc`).
- `EOF` si la entrada terminó **antes** de leer nada.

Por eso, **siempre** se compara el retorno con lo esperado. Es la única forma de saber si salió bien.

5.3 `%c`, `%s` y `%[...]`: tres formas muy distintas de leer texto

Formato	Qué lee	¿Salta espacios?	Nota
`%c`	**un** carácter	**No**	Si el próximo byte es espacio o `\n`, **ese** te da
`%s`	una **palabra** (sin espacios)	**Sí**	**Peligroso**: sin ancho, puede desbordar
`%[...]`	los caracteres **de un conjunto**	**No**	`%[^\n]` = "todo hasta el `\n`"

Ejemplos de cómo se comportan ante la entrada `hola mundo`:

- `scanf("%c", &c)` → te da el **primer espacio**, no la `h`.
- `scanf("%s", word)` → salta el espacio y te da `hola` (se detiene en el espacio).
- `scanf("%[^\n]", line)` → te da `hola mundo` **completo** (lee todo menos el `\n`).

> ***El espacio en el formato.*** *Un espacio en la cadena de formato significa "salteen cero o más espacios". Por eso* `" %c"` *(con espacio* ***antes*** *de* `%c`*) salta el* `\n` *pendiente y recién ahí lee el carácter. Es el truco para leer un* `%c` *después de un* `%d`*.*

5.4 Los dos superpoderes (y una advertencia): ancho y `*`

Ancho máximo. Un número entre `%` y la letra limita cuánto lee: `%19s` lee **como máximo 19** caracteres (más el `'\0'`), para un búfer de 20. **Nunca** uses `%s` sin ancho: es el peligro de desbordar.

```c
char word[20];
scanf("%19s", word);     /* a lo sumo 19 letras + '\0': seguro */
```

Supresión (`*`). Un `*` después del `%` significa "leé esto, pero **no lo guardes**". Sirve para saltear prefijos:

```c
int value;
scanf("%*s %d", &value);   /* lee y TIRA una palabra; guarda solo el número */
/* ante "ID: 12345", descarta "ID:" y guarda 12345 */
```

La advertencia (`%n`). `%n` **no lee**: guarda cuántos caracteres se leyeron hasta ahí. Sirve para saber "por dónde quedó", pero se lo considera riesgoso: no lo uses con formatos que vengan de afuera.

5.5 El peligro del tipo equivocado

`scanf` **no sabe** qué tipo le pasaste: se guía **solo** por el formato. Si el formato dice `%d` (espera un `int *`) y le pasás un `double *`, escribe un `int` donde vos esperabas un `double`: **comportamiento indefinido**. Puede dar basura, corromper memoria, o "andar" por casualidad hasta que deje de andar.

> ***Regla de oro de `scanf`:*** *el formato y el puntero* ***tienen que coincidir exactamente****. El compilador no puede revisarlo (por los `...`), así que la responsabilidad es toda tuya. Ojo con el clásico: en `scanf`, para un `double` se usa `%lf` (¡no `%f`, como en `printf`!).*

5.6 Falta `%i`, `%u`, `%x` … y `gets` ya no existe

- `%i` adivina la base (decimal, `0x` hex, `0` octal); `%d` solo decimal.
- `%u` es para enteros **sin signo**; `%f` para `float`; `%lf` para `double`.
- `gets` **fue eliminada** del estándar (C11) porque leía una línea **sin límite** de tamaño: no podía saber cuándo parar y **desbordaba** el búfer. Su reemplazo seguro es `fgets` **con tamaño**.

5.7 `sscanf` : el mismo `scanf` , pero sobre una cadena

`sscanf` es idéntico a `scanf` , salvo que **no lee de un flujo** (`stdin`), sino de una **cadena** que ya tenés en memoria. Es el compañero natural de `fgets` : leés la línea **una vez** y después la interpretás sin volver a tocar `stdin` .

```c
#include <stdio.h>

int main(void) {
    char line[100];
    if (fgets(line, sizeof line, stdin) == NULL) {
        return 0;
    }

    int a, b;
    /* sscanf(cadena, formato, direcciones...):
       igual que scanf, pero la "fuente" es `line` en vez de stdin.
       Devuelve cuántos valores asignó (2 si leyó los dos enteros). */
    if (sscanf(line, "%d %d", &a, &b) == 2) {
        printf("suma: %d\n", a + b);
    }

    return 0;
}
```

> ***Por qué conviene.*** *Leer **una vez** con `fgets` y después analizar con `sscanf` es más seguro que encadenar `scanf` sobre `stdin` : controlás el tamaño de la línea, la ves completa y, si el formato no cierra, **no** dejaste residuos en el flujo.*

Práctica de la Parte 5

Ejercicio 13.10 — Validador de edad numérica

Conceptos · `scanf` , valor de retorno, fallo de conversión, descarte de entrada inválida

Contexto · Los formularios tienen que sobrevivir a que alguien escriba texto donde se pide un número. Simplificación: pedir una edad y detectar el error sin comportarse de forma errática.

Consigna · Pedir la edad con `scanf("%d", &age)`. Si la conversión **falla** (el retorno **no** es `1`), mostrar un mensaje de entrada inválida y **no** usar `age`. El programa **no** debe bloquearse ni leer datos basura.

Entrada · Una línea con un número (caso válido) o con texto (caso inválido). **Salida** · La edad registrada, o el mensaje de error.

Escenario de prueba

```
Entrada:  34
Salida:   Edad registrada: 34

Entrada:  treinta
Salida:   Entrada inválida: se esperaba un número entero.
```

Criterios de aceptación

- ☐ Comprueba `scanf(...) == 1`.
- ☐ Con texto, informa el error y **no** usa un valor no inicializado.
- ☐ Con un número válido, lo muestra correctamente.
- ☐ No deja la entrada "sucia" para una lectura posterior.

Ejercicio 13.11 — Extracción de tokens con formato avanzado

Conceptos · `scanf` con ancho máximo (`%19s`), supresión (`*`), valor de retorno

Contexto · Muchas líneas traen etiquetas (`ID: 12345`) y solo interesa el valor. Simplificación: extraer el número descartando el prefijo, con un solo `scanf`.

Consigna · Leer una línea con etiqueta y **extraer solo el valor numérico**, descartando el prefijo textual. Usá un **ancho máximo** para el token del prefijo y el modificador `*` para que `scanf` **no lo guarde**.

Entrada · Una línea con formato `ETIQUETA: número`. **Salida** · El valor numérico con una etiqueta clara.

Escenario de prueba

```
Entrada:  ID: 12345
Salida:   valor = 12345

Entrada:  REF: 00042
Salida:   valor = 42
```

Criterios de aceptación

- [ ] Usa el ancho máximo en el token descartado (`%19s` o similar).
- [ ] Usa `*` para **no** asignar la etiqueta.
- [ ] Extrae el número correctamente en ambos escenarios.
- [ ] Comprueba el retorno de `scanf` (`== 1`).

Ejercicio 13.12 — La calculadora de una línea

Conceptos · `scanf` con formato mixto (`%d %c %d`), valor de retorno

Contexto · Una calculadora de bolsillo acepta `número operador número` . Simplificación: leer una cuenta de suma o resta con un solo `scanf` .

Consigna · Leer una línea con el formato `número operador número` (por ejemplo `8 + 5`) usando `scanf("%d %c %d", &a, &op, &b)` . Comprobar que se asignaron **3** valores. Si el operador es `+` , mostrar la suma; si es `-` , la resta; si es otro, avisar que no se reconoce.

Entrada · Una cuenta de una línea. **Salida** · El resultado, o el aviso de operador no reconocido.

Escenario de prueba

```
Entrada:  8 + 5
Salida:   Resultado: 13

Entrada:  10 - 4
Salida:   Resultado: 6

Entrada:  3 x 4
Salida:   Operador no reconocido: 'x'
```

Criterios de aceptación

- [ ] Usa `scanf("%d %c %d", ...)` y comprueba que el retorno sea `3`.
- [ ] Resuelve suma y resta correctamente.
- [ ] Avisa en el caso de operador no reconocido.
- [ ] No se rompe si la línea no trae los tres datos.

PARTE 6 — Caracteres: cuando "una letra" pesa más de un byte

Hasta acá contamos bytes. Pero, ¿un byte es siempre "una letra"? Spoiler: no.

6.1 Byte no es lo mismo que carácter

- Un **byte** es una unidad de **almacenamiento**: `sizeof(char)` es siempre `1`, y en la práctica son 8 bits. Es una medida de **cuánto ocupa**.
- Un **carácter** es una noción **humana**: "la letra `á`", "el símbolo `€`". Es una medida de **significado**.

La relación entre ambos la define la **codificación**: cuántos bytes (y con qué valores) representan cada carácter.

6.2 `EOF`, `'\0'`, `'\n'` y `'\r'`: cuatro parecidos que no se parecen

Antes de seguir, despejemos cuatro nombres que se confunden siempre:

Símbolo	Qué es	Valor	¿Es un byte de dato?
`EOF`	"se acabó la entrada"	`-1`	**No** (es una señal)
`'\0'`	carácter **nulo**, terminador de cadenas	`0`	Sí
`'\n'`	fin de línea (**LF**)	`10` (`0x0A`)	Sí
`'\r'`	retorno de carro (**CR**)	`13` (`0x0D`)	Sí

6.3 ASCII, UTF-8 y la `á` de dos bytes

ASCII es la codificación clásica: 128 caracteres (letras inglesas, dígitos, signos), **todos de 1 byte**. `'A'` es `65`, `'\n'` es `10`.

UTF-8 es la codificación de hoy (Linux, macOS, la web) y es de **longitud variable**:

- Los caracteres **ASCII** ocupan **1 byte** (compatibilidad total).
- Los demás ocupan **2, 3 o 4 bytes**. El primer byte "avisa" cuántos siguen.

Carácter	Bytes (hex)	¿Cuántos?
`A`	`41`	1 byte
`ñ`	`C3 B1`	2 bytes
`á`	`C3 A1`	2 bytes
`€`	`E2 82 AC`	3 bytes
😜	`F0 9F 98 80`	4 bytes

6.4 Por qué esto te importa de verdad

Como `getchar` entrega **bytes** (no glifos), leer un texto con acentos tiene consecuencias concretas:

- Si tecleás `á`, `getchar` te va a dar **dos** bytes (`0xC3` y luego `0xA1`). Dos llamadas, **un** símbolo.
- `strlen("á")` da **2**, no `1`.
- Si recorrés "carácter por carácter" con `getchar`, vas a **partir** los multibyte al medio.

Reglas prácticas:

1. **Separá las capas.** ¿Querés contar **bytes** o **símbolos**? No es lo mismo.
2. **Cuando examinés bytes crudos, usá `unsigned char`.** Comparar `0xC3` contra un `char` con signo trae resultados confusos.
3. **Un `char` no alcanza para todo símbolo.** Para eso existen `wchar_t` y `char32_t` (capítulo `15-tipos-stddef`).
4. **No adivines la codificación.** Si los acentos salen "raros", casi siempre es que la codificación del texto no coincide con la que espera la terminal.

Práctica de la Parte 6

Ejercicio 13.13 — Filtro de terminadores de línea

Conceptos · `getchar`, `EOF`, `'\n'`, `'\r'`, `'\0'`, distinción de símbolos

Contexto · Los analizadores de texto necesitan distinguir los símbolos de control de los caracteres comunes. Simplificación: etiquetar cada símbolo especial.

Consigna · Leer de `stdin` byte a byte y, por cada símbolo:

- `'\n'` → imprimir `<LF>`;
- `'\r'` → imprimir `<CR>`;
- `'\0'` → imprimir `<NUL>`;
- cualquier otro → imprimir el carácter tal cual;
- al llegar a `EOF` → imprimir `<EOF>` y terminar.

Entrada · Secuencia de bytes hasta `EOF`. **Salida** · El carácter (o su etiqueta) por cada símbolo; `<EOF>` al final.

Escenario de prueba

```
printf 'a\nb\rc\0d' | ./filtro
a
<LF>
b
<CR>
c
<NUL>
d
<EOF>
```

> `'\0'` *solo aparece si la entrada es un flujo con bytes* `0x00` *(por ejemplo, una tubería); no se puede teclear en una terminal.*

Criterios de aceptación

- [ ] Distingue `'\n'`, `'\r'`, `'\0'` y `EOF` correctamente.
- [ ] Imprime el resto de los caracteres sin alterarlos.
- [ ] Detecta `'\0'` como byte real `0x00` (no lo confunde con `EOF`).
- [ ] Termina en el primer `EOF`, con la etiqueta `<EOF>` incluida.

Ejercicio 13.14 — Radiografía de un acento

Conceptos · `getchar`, bytes altos, UTF-8 multibyte, `unsigned char`

Contexto · Ver "por dentro" cómo se guarda una letra acentuada es la mejor forma de entender multibyte. Simplificación: mostrar cada byte de la entrada en hexadecimal.

Consigna · Leer de `stdin` byte a byte y mostrar cada uno en **hexadecimal** (por ejemplo `C3 A1`), separados por espacio, hasta `EOF`. Al final, mostrar también el **total de bytes**.

Entrada · Cualquier texto hasta `EOF` (probá con `á`, `ñ`, `€`). **Salida** · La seguidilla de bytes en hex y el total.

Escenario de prueba

```
printf 'á' | ./radio
Bytes: C3 A1
Total: 2 bytes
```

```
printf 'A' | ./radio
Bytes: 41
Total: 1 byte
```

> ***Lo que tenés que ver:*** *`á` son **dos** bytes (`C3 A1`), `A` es **uno** (`41`). Ahí está el multibyte, en la mano.*

Criterios de aceptación

- [ ] Lee con `getchar` en un `int` y corta ante `EOF`.
- [ ] Muestra cada byte en hexadecimal, con dos dígitos (por ejemplo `0A`, no `A`).
- [ ] Informa el total de bytes leídos.
- [ ] El acento da **2** bytes y la `A` da **1** byte.

PARTE 7 — Robustez: que no se rompa nunca

Un programa frágil **asume** que la entrada es perfecta. Un programa robusto **asume lo contrario**. Esta parte es corta pero salvavidas.

7.1 Comprobá siempre los retornos

Cada lectura te **avisa** cómo salió. Ignorar el aviso es pedir un desastre. Este es el reflejo que tenés que automatizar:

```
int c;
while ((c = getchar()) != EOF) { /* ... */ }        /* getchar contra EOF          */

char line[100];
if (fgets(line, sizeof line, stdin) == NULL) {       /* fgets contra NULL           */
    /* fin de entrada o error */
}

int n;
if (scanf("%d", &n) != 1) {                          /* scanf contra lo esperado    */
    /* no se leyó un número */
}
```

7.2 `feof` y `ferror`: ¿se acabó o se rompió?

Cuando una lectura falla, puede ser por **dos** motivos distintos: se acabó la entrada (**normal**) o hubo un **error** (falla real). Dos funciones los separan:

```
int feof(FILE *stream);     /* ¿se llegó al final de la entrada? */
int ferror(FILE *stream);   /* ¿hubo un error de lectura/escritura? */
```

Y `perror` imprime un mensaje legible del último error, en `stderr`:

```
if (ferror(stdin)) {
    perror("stdin");        /* imprime algo como: stdin: <motivo> */
}
```

`clearerr(stream)` **borra** los indicadores de error y de fin de archivo. Se usa si querés **seguir leyendo** después de haber tocado el final (si no, el flujo "queda pegado" en `EOF`).

7.3 El bug estrella: el bucle infinito

Si la entrada **termina** y tu bucle **no** corta, vuelve a leer, recibe `EOF` de nuevo, y **gira para siempre**. El patrón correcto es **siempre** tener una condición de corte:

```c
int c;
while ((c = getchar()) != EOF) { /* procesar c */ }

/* Con scanf: cortar si no asignó lo esperado. */
int n;
while (scanf("%d", &n) == 1) { /* procesar n */ }
```

> ***Caso especial.*** *Si* `scanf` *falla porque el* ***formato no coincide****, deja el carácter culpable* ***sin consumir****. Si no lo descartás, el bucle lo vuelve a encontrar y* ***no avanza****: bucle infinito. La solución es descartar el carácter problemático (o el resto de la línea) o abandonar la lectura.*

7.4 La entrada puede ser maliciosa

Si la entrada viene de cualquiera, no confíes:

- **Acotá siempre** los búferes (nunca `%s` sin ancho, nunca `gets`).
- **Comprobá rangos**: que un entero entre en el tipo, que un índice sea válido.
- **Convertí con control de error**, en vez de confiar en conversiones "optimistas".

7.5 `strtol` y `strtod`: convertir texto a número sin sorpresas

`atoi` y `atof` **no avisan** cuando la conversión falla: devuelven `0` y listo. Para validar en serio, `strtol` (texto → `long`) y `strtod` (texto → `double`) te dan **tres** señales en una sola llamada:

```c
#include <stdlib.h>   /* strtol, strtod */
#include <errno.h>    /* errno          */

/* strtol(cadena, endptr, base):
   - Devuelve el valor convertido.
   - endptr (puntero al puntero): queda apuntando al primer carácter NO convertido.
       · si endptr == inicio           → no se convirtió nada (no era un número);
       · si *endptr == '\0'            → se consumió toda la cadena (todo era el
número);
       · si *endptr apunta a otra cosa  → sobra basura después del número.
   - errno se pone en ERANGE si el valor no entra en un long.
     Por eso se limpia errno = 0 ANTES de llamar. */
char *endptr;
errno = 0;
long value = strtol(text, &endptr, 10);
```

Con esas tres señales (¿convirtió algo?, ¿sobró basura?, ¿hubo desborde?) tenés todo para decidir si la entrada es válida. `strtod` funciona igual, pero para `double`.

Práctica de la Parte 7

Ejercicio 13.15 — Bucle robusto ante cierre de terminal

Conceptos · `getchar`, `EOF`, corte de bucles, fin de entrada por teclado

Contexto · Un programa interactivo tiene que terminar **limpio** cuando el usuario cierra la entrada, sin quedar girando. Simplificación: contar caracteres y líneas hasta `EOF`.

Consigna · Leer de `stdin` con `getchar()` hasta `EOF`. Contar los caracteres y las líneas (cantidad de `'\n'`). Al detectar `EOF`, mostrar el resumen y un mensaje de cierre. **No** debe haber bucle infinito.

Entrada · Líneas de texto, cerradas con **Ctrl+Z** (Windows) o **Ctrl+D** (Linux/macOS). **Salida** · Conteo de caracteres y líneas, y mensaje de fin.

Escenario de prueba

```
Entrada:
hola
mundo
(Ctrl+Z + Enter  /  Ctrl+D)

Salida:
Leídos 11 caracteres en 2 líneas.
Fin de la entrada (EOF). Chau.
```

> `hola\n` *son 5 caracteres y* `mundo\n` *son 6:* ***11*** *en total y* ***2*** *saltos de línea.*

Criterios de aceptación

- [ ] El bucle corta ante `EOF` (no hay giro infinito).
- [ ] Cuenta caracteres y líneas correctamente.
- [ ] Con la entrada del ejemplo, informa **11** caracteres y **2** líneas.
- [ ] Muestra un mensaje de cierre y termina.

Ejercicio 13.16 — Validador de número a prueba de balas

Conceptos · `fgets`, `strtol`, validación de rango, detección de basura

Contexto · Un dato confiable no es solo "un número": es un número **dentro de un rango** y **sin basura alrededor**. Simplificación: leer un puerto de red (0 a 65535) y validarlo del todo.

Consigna · Leer una línea con `fgets` y convertirla con `strtol`. La entrada es válida **solo si**: se convirtió algo, **no** sobra nada (salvo el `'\n'`), y el valor está entre `0` y `65535`. Si algo falla, informar el motivo exacto.

Entrada · Una línea de texto. **Salida** · El puerto validado, o el motivo del rechazo.

Escenario de prueba

```
Entrada:  8080
Salida:   Puerto válido: 8080

Entrada:  8080x
Salida:   Error: hay caracteres extra después del número ('x')

Entrada:  70000
Salida:   Error: 70000 está fuera del rango 0..65535

Entrada:  puerto
Salida:   Error: 'puerto' no es un número
```

Criterios de aceptación

- [ ] Lee la línea con `fgets` y comprueba el retorno.
- [ ] Usa `strtol` con el puntero de fin (`endptr`) para saber hasta dónde convirtió.
- [ ] Detecta: basura antes, basura después, vacío y fuera de rango.
- [ ] Un valor válido se acepta; los cuatro escenarios de error coinciden.

PARTE 8 — Proyecto final: tu propia consola de comandos

Llegó la hora de juntar **todo**. Vas a construir un intérprete de comandos **mínimo**: una versión de juguete de tu terminal.

8.1 Qué vas a construir

Un programa que muestre un **prompt** `>`, espere una orden y reaccione. No va a ejecutar programas reales: solo va a entender unos pocos comandos. Pero **todo** lo que aprendiste va a estar adentro: leer líneas acotadas, interpretar el comando, validar números, manejar residuos del búfer y cerrar con `EOF` o con `salir`.

8.2 Los comandos a entender

Orden	Qué hace
`eco <texto>`	Imprime `<texto>` tal cual (puede tener espacios).
`suma <a> <b>`	Imprime la suma de dos enteros.
`salir`	Termina el programa.
(otra cosa)	Error de comando desconocido.

Ejercicio final

Ejercicio 13.17 — (JEFE) Simulador de consola de comandos interactiva

Conceptos · `fgets`, `sscanf`, `stdin` / `stdout` / `stderr`, limpieza de búfer, validación numérica, `EOF`, corte de bucles, comandos con formato.

Contexto · Un intérprete de comandos (un "shell" mínimo) es el ejercicio que reúne **todo** el capítulo: lee órdenes, interpreta argumentos, valida errores sin bloquearse y termina limpio. Es una **simplificación** de una consola real: no ejecuta programas, solo reconoce unos pocos comandos.

Consigna · Desarrollar una aplicación de línea de comandos que:

1. Muestre un **prompt** `>` y lea órdenes del usuario combinando `fgets` y `sscanf` según el comando.
2. Reconozca los comandos de la tabla de 8.2 (`eco` , `suma` , `salir`).
3. Maneje correctamente los **residuos de búfer** cuando la entrada **excede el tamaño esperado** (comando demasiado largo, o una línea con más datos de los que consume el formato).
4. **Valide** errores de formato numérico (por ejemplo, `suma 4 abc`) **sin bloquearse**.
5. Termine de forma limpia al detectar `EOF` o al recibir `salir` , **sin** bucles infinitos ni comportamiento indefinido.

Entrada · Órdenes por `stdin` , una por línea; la sesión cierra con `salir` o `EOF` . **Salida** · El resultado de cada comando por `stdout` ; los errores por `stderr` ; un cierre al final.

Escenario de prueba

```
Entrada:
eco hola mundo
suma 4 7
suma 4 abc
xyz
salir

stdout:
> hola mundo
> 11
> Error: 'abc' no es un número válido
> Error: comando desconocido: xyz
> Chau.
```

```
Entrada:
(Ctrl+Z + Enter  /  Ctrl+D)

stdout:
> Fin de entrada (EOF). Chau.
```

Caso de orden demasiado larga (con búfer de 128 bytes):

```
Entrada:  eco <200 caracteres>
Salida:   Error: la orden supera el tamaño máximo del búfer.
```

Identificadores sugeridos · `line`, `command`, `arg1`, `arg2`, `LINE_SIZE`, `matched`, `running`

Restricciones/pistas

- Leer cada orden con `fgets` en un búfer **acotado** y detectar truncamiento (3.4).
- Para `suma`, interpretar los dos enteros con `sscanf` sobre la línea ya leída (más seguro que `scanf` directo sobre `stdin`) y **comprobar que devolvió `2`**; si no, error de formato.
- Descartar el resto de la línea cuando haga falta (4.5). **No** usar `fflush(stdin)`.
- Cortar ante `EOF`: `fgets` devuelve `NULL`.
- Separar la salida normal (`stdout`) de los errores (`stderr`).

Criterios de aceptación

- ☐ Muestra el prompt y lee órdenes con `fgets` acotado.
- ☐ `eco` imprime el texto conservando los espacios internos.
- ☐ `suma` con dos enteros válidos imprime la suma correcta.
- ☐ `suma` con un argumento no numérico informa el error y **continúa** (no se bloquea).
- ☐ Un comando desconocido informa el error y **continúa**.
- ☐ Una orden que excede el búfer se detecta y no corrompe la memoria ni la lectura siguiente.
- ☐ `salir` y el `EOF` terminan el programa limpiamente (con mensaje de cierre).
- ☐ No hay bucles infinitos ni comportamiento indefinido ante entradas inesperadas.
- ☐ Errores por `stderr`, resultados por `stdout`.

Cierre: lo que te llevás

Si tuvieras que quedarte con **una sola idea**, que sea esta: **leer de la consola es leer de una tubería con búfer**. Todo lo demás se deduce de ahí.

- Tus programas tienen **tres tuberías** (`stdin` , `stdout` , `stderr`), y el **shell** decide a qué está conectada cada una. Tu código no cambia.
- **Elegí la función por su trabajo**: `getchar` (un byte), `fgets` (una línea, seguro), `scanf` (con formato, cómodo pero resbaladizo).
- `EOF` **es una señal** (un `int` negativo), no un carácter. Nunca lo guardes en un `char` .
- **El búfer causa los misterios**: `scanf` deja el `\n` pendiente; por eso existe la limpieza de línea, y por eso `fflush(stdin)` está prohibido.
- **La robustez es una decisión**: comprobar retornos, acotar búferes, validar datos y cortar ante `EOF` .
- **Un carácter no siempre es un byte**: una `á` son dos bytes.

Y el puente hacia lo que viene: **para C, un archivo es un flujo más**. En `14-archivos` vas a reusar **exactamente** estas ideas — `fgetc` , `fgets` , `fscanf` , `feof` , los búferes, `EOF` — solo que sobre un flujo abierto con `fopen` en lugar de sobre `stdin` . Si dominaste este capítulo, el de archivos es casi el mismo capítulo con una apertura y un cierre.