Capítulo 13 — Argumentos de línea de comandos: de cero a pro

> *Capítulo intermedio, pero **escrito para que se entienda de verdad**. Se apoya en variables (`02`), operadores (`03`), `if` y bucles (`04`), funciones (`05`), arreglos y cadenas (`06`) y punteros (`07`).*
>
> *Es la **antesala** del capítulo de entrada/salida estándar (`14-entrada-salida-estandar`). Acá aprendés a que tu programa reciba **órdenes al nacer** (cuando lo ejecutás) en lugar de pedirlas recién mientras corre. En el capítulo siguiente vas a profundizar en las tres tuberías (`stdin` / `stdout` / `stderr`), que acá tocamos apenas lo justo.*

Cómo leer este capítulo

Hasta ahora, todos tus programas empezaban **preguntando**. Le mostraban un cartel al usuario y esperaban a que escribiera algo por teclado. Eso funciona, pero no es la única forma. Las herramientas serias — `ls` , `grep` , `git` , `gcc` — no te preguntan nada: **les pasás las órdenes en el mismo momento en que las ejecutás**.

Ese es el tema de este capítulo.

Está dividido en **nueve 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	Órdenes al nacer	Tu programa puede recibir los datos cuando lo ejecutás, no después.
2	`argc` y `argv`	El par mágico que `main` recibe con la cantidad y la lista de argumentos.
3	`argv[0]`	El primer argumento siempre es **cómo llamaron** a tu programa.
4	Recorrer y clasificar	Cómo leer los argumentos uno por uno y distinguir qué tipo es cada uno.
5	Opciones cortas y largas	`-v` vs `--verbose`: dos estilos que conviven en toda herramienta.
6	Opciones con valor	`--dato=2` o `--dato 2`: cómo leer el valor que acompaña a una opción.
7	Validar y avisar	Errores por `stderr`, códigos de salida y un `--help` digno.
8	`getopt`	El parseo ordenado de opciones cortas, con soporte para `-abc` apiladas.
9	`getopt_long`	El nivel pro: opciones largas al estilo `--output=archivo`.

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 puede recibir órdenes al nacer

Antes de tocar una sola línea de código, entendamos **qué cambia** y **por qué** esto es tan útil.

1.1 La diferencia entre preguntar y que te digan

Pensá en una **máquina expendedora**.

- **Forma antigua (preguntar durante la ejecución):** prendés la máquina y ella te va preguntando: *"¿Qué querés?"*. Vos respondés por un teclado mientras la máquina ya está funcionando. En C, eso es usar `scanf`, `getchar` o `fgets` sobre `stdin`: el programa **se frena y espera**.
- **Forma nueva (recibir órdenes al nacer): le das los ingredientes ANTES de encenderla**. Le decís: *"2 latas de Coca-Cola"*. La ejecutás y ya sabe qué hacer: **no pregunta, va al resultado**.

Eso es exactamente lo que pasa cuando escribís en la terminal:

```
./programa test.txt --alguna-opcion --dato=2
```

Le estás pasando **órdenes al programa en el mismo momento en que lo ejecutás**. Son los **argumentos de línea de comandos** (en inglés, *command-line arguments*).

> ***Momento ajá.*** *La diferencia clave es el **momento**: los argumentos llegan **al arrancar el programa** (cuando empieza `main`), no durante su ejecución.*

1.2 ¿Por qué esto importa?

Porque es **el** mecanismo con el que se automatiza trabajo en la vida real.

- Un script de respaldo hace `./backup --origen=/data --destino=/backup --comprimir`.
- Un analizador de logs hace `./analiza server.log --desde=2026-01-01 --nivel=error`.
- El compilador hace `gcc -o salida programa.c -Wall`.

Nadie está sentado respondiendo preguntas: los datos **viajan con la orden**. Eso permite que un programa llame a otro, encadenarlos con tuberías y automatizarlos con tareas programadas.

1.3 Lo que ya conocés, y lo que viene

Ya sabés que `main` es la puerta de entrada del programa. Su forma mínima es:

```c
int main(void)
{
    /* ... */
    return 0;
}
```

Ese `int` de adelante es el **código de retorno**: `0` significa "salió bien". Ya lo veníamos usando.

Para recibir argumentos, `main` admite **dos parámetros extra**. Esa es toda la novedad:

```c
int main(int argc, char *argv[])
{
    /* ... */
    return 0;
}
```

No pasa nada raro: es la misma función `main`, con dos datos de entrada más. En la Parte 2 los desarmamos pieza por pieza.

Práctica de la Parte 1

Ejercicio 13.1 — Pensar antes de ejecutar

Conceptos · argumentos de línea de comandos, diferencia con la lectura por teclado

Contexto · Antes de escribir código, hay que entender **cuándo** llegan los datos. Simplificación: dos formas de pedirle algo a un programa.

Consigna · Leé estas dos escenas y respondé por escrito (sin programar):

1. Ejecutás `./programa`, aparece el cartel `¿Archivo?` y vos escribís `datos.txt`.
2. Ejecutás `./programa datos.txt` directamente desde la terminal.

¿En cuál de las dos el programa **ya sabe qué archivo usar desde el arranque**? ¿Cuál corresponde a "argumentos de línea de comandos"?

Entrada · Ninguna (es un ejercicio de análisis). **Salida** · Una respuesta escrita comparando ambas formas.

Escenario de prueba · No aplica (no hay programa).

Criterios de aceptación

- [ ] Identificás que en la escena 2 los datos llegan **al ejecutar**.
- [ ] Explicás la diferencia entre pedir datos **durante** la ejecución y **al ejecutarla**.
- [ ] Nombrás correctamente la forma 2 como "argumentos de línea de comandos".

PARTE 2 — `argc` y `argv`: el par mágico

2.1 `argc`: el contador

Cuando ejecutás un programa desde la consola, el **sistema operativo** (a través del shell) le entrega a `main` **cuántas** palabras escribiste y **cuáles** fueron.

El primero de los dos parámetros es `argc`:

- Significa **"argument count"**: *cantidad de argumentos*.
- Es un `int`.
- Cuenta **cuántas palabras** separadas por espacios llegaron en la línea de comandos.

Analogía. Imaginá una lista de compras escrita en un papel. `argc` es **el número de ítems** del papel. No te dice qué comprar: solo **cuántas cosas** hay.

> ***Para guardar.*** `argc` ***nunca es 0*** *en un programa ejecutado normalmente. Siempre cuenta al menos* ***1****. ¿Por qué? Lo vas a ver en la Parte 3 (spoiler: el nombre del programa también cuenta).*

2.2 `argv` : la lista

El segundo parámetro es `argv` :

- Significa **"argument vector"**: *vector (lista) de argumentos*.
- Es un **arreglo de cadenas**: `char *argv[]` .
- Cada posición guarda **una palabra** exactamente como apareció en la terminal.

Analogía. Si `argc` es el número de ítems de la lista, `argv` es **la lista misma**: un estante donde cada balda guarda una palabra completa. `argv[0]` es el primer estante, `argv[1]` el segundo, y así hasta `argv[argc-1]` .

> ***Un detalle jurídico.*** *`argv` es un arreglo de **punteros a** `char` (`char *`). Cada elemento apunta al comienzo de una cadena terminada en `'\0'` . Por eso se escribe `char *argv[]` y no `char argv[]` : no guardamos las letras, guardamos **la dirección** de dónde empiezan. Si todavía tenés dudas con punteros, repasá el capítulo `07-punteros` .*

2.3 El ejemplo completo, palabra por palabra

Si ejecutás:

```
./programa test.txt --alguna-opcion --dato=2
```

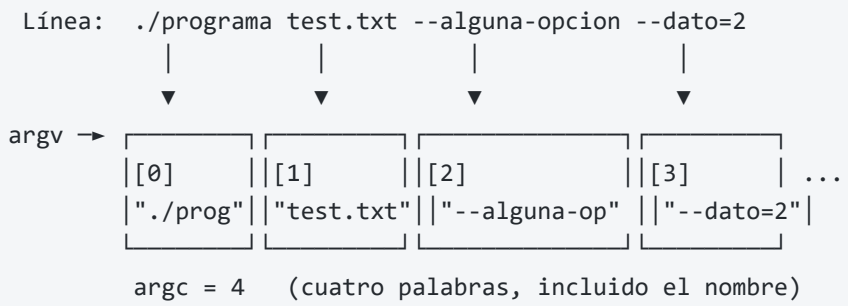
el sistema reparte los argumentos así:

Índice	Valor de `argv[i]`	¿Qué es?
`0`	`"./programa"`	El nombre con el que se ejecutó.
`1`	`"test.txt"`	Un argumento pasado.
`2`	`"--alguna-opcion"`	Otro argumento.
`3`	`"--dato=2"`	Otro más.

Y `argc` vale **4** (hay 4 palabras). Fijate que `argc` **cuenta también el nombre**: por eso `argv[0]` a `argv[3]` son 4 posiciones, y `argc = 4` .

> ***Ojo con esto.*** *El* ***espacio*** *es lo que separa un argumento del siguiente. Cada bloque entre espacios se convierte en un* `argv[i]` *distinto. Si querés que un texto con espacios sea* ***un solo*** *argumento, hay que* ***encerrarlo entre comillas****:* `./programa "hola mundo"` *llega como un único* `argv[1]` *igual a* `"hola mundo"` *.*

2.4 Diagrama mental



2.5 El caso más chico posible

Si ejecutás:

```
./saludo
```

obtenés:

- `argc = 1`
- `argv[0] = "./saludo"`

Aunque **no pasaste nada extra**, `argc` vale 1: el nombre del programa **siempre** ocupa el estante 0.

Práctica de la Parte 2

Ejercicio 13.2 — Contar y adivinar

Conceptos · `argc`, `argv`, índice de argumentos

Contexto · Antes de programar, hay que **leer** una línea de comandos como la leería el sistema. Simplificación: adivinar valores sin ejecutar nada.

Consigna · Dada la invocación:

```
./programa archivo.txt -v 42
```

respondé sin programar:

1. ¿Cuánto vale `argc`?
2. ¿Qué contiene `argv[0]`, `argv[1]`, `argv[2]` y `argv[3]`?
3. ¿Por qué `argc` **no** vale 3?

Entrada · La línea de comandos del enunciado. **Salida** · Las tres respuestas.

Escenario de prueba · No aplica (no hay programa).

Criterios de aceptación

- [ ] `argc` correcto (recordá que el nombre cuenta).
- [ ] Los cuatro valores de `argv` correctos y en orden.
- [ ] Justificás el punto 3 mencionando que `argv[0]` es el nombre del programa.

Ejercicio 13.3 — Eco de argumentos

Conceptos · `argc`, `argv`, bucle `for`, `%s`

Contexto · El primer programa "de verdad" con argumentos: repetir lo que recibió. Simplificación: un eco que no interpreta nada, solo muestra.

Consigna · Escribí un programa que, al ejecutarlo, muestre por pantalla **la cantidad** de argumentos y luego **cada argumento** en una línea, en el formato del escenario de prueba. Recorré el arreglo con un bucle `for`.

Entrada · Los argumentos que le pases al ejecutarlo. **Salida** · Una línea con la cantidad y una línea por argumento, con el formato exacto del ejemplo.

Escenario de prueba

```
$ ./eco hola mundo 123
argc = 4
argv[0] = ./eco
argv[1] = hola
argv[2] = mundo
argv[3] = 123
```

Identificadores sugeridos · `argc` , `argv` , `i`

Restricciones / pistas

- El bucle debe llegar **hasta** `argc-1` : nunca leas `argv[argc]` (estarías fuera del arreglo).
- Empezás en `i = 0` si querés mostrar **todos** (incluido el nombre). En el próximo ejercicio empezarás en `i = 1` .

Criterios de aceptación

- ☐ Muestra el valor correcto de `argc` .
- ☐ Recorre de `0` a `argc-1` con un `for` .
- ☐ El formato coincide exactamente con el escenario de prueba.
- ☐ No accede a ninguna posición fuera del rango válido.

PARTE 3 — `argv[0]` : el nombre del programa

3.1 El argumento número cero es especial

`argv[0]` guarda **cómo se llamó al programa** en esa ejecución concreta. No es un dato que pasaste vos: es el nombre con el que el sistema lo invocó.

Analogía. Es como presentarte: *"Hola, soy Horacio"*. `argv[0]` es **el nombre con el que te llamaron** en ese momento, no quién sos en realidad.

Puede ser una ruta relativa (`./programa`), absoluta (`/usr/bin/programa`) o, en Windows, incluir `.exe` (`programa.exe`). Todo depende de **cómo lo escribiste al ejecutarlo**.

3.2 ¿Para qué sirve saber el nombre?

Para **mensajes honestos**.

Cuando un programa necesita decirle al usuario cómo usarlo (un mensaje de "Uso: ..."), lo correcto es mostrar **el mismo nombre con el que lo llamaron**, no uno fijo escrito a mano.

- Si el usuario lo ejecutó como `./mini` , tu ayuda debería decir `Uso: ./mini ...` .
- Si lo ejecutó como `mini` , debería decir `Uso: mini ...` .

Nunca hardcodees el nombre del ejecutable. Usá `argv[0]` .

> ***Momento ajá.*** *El truco* `printf("Uso: %s <archivo>\n", argv[0]);` *es uno de los patrones más repetidos de la programación de consola. Guardátelo.*

3.3 Un caso de borde

`argv[0]` **puede estar vacío** (`""`) en situaciones muy raras de algunos sistemas, pero en la práctica casi nunca ocurre. Lo normal es que tenga el nombre, relativo o absoluto.

Práctica de la Parte 3

Ejercicio 13.4 — El nombre con el que me llamaron

Conceptos · `argv[0]` , cadenas, `printf` con `%s`

Contexto · Todo programa de consola debería mostrar su propio nombre en las ayudas. Simplificación: mostrar el nombre y una línea de uso genérica.

Consigna · Escribí un programa que muestre su nombre (`argv[0]`) y un mensaje de uso que lo incluya **dinámicamente**, con el formato del escenario de prueba.

Entrada · Cualquier cosa (el programa no necesita argumentos extra). **Salida** · Dos líneas con el formato exacto del ejemplo.

Escenario de prueba

```
$ ./demo archivo.txt
Programa: ./demo
Uso: ./demo <archivo> [opciones]
```

Identificadores sugeridos · `argv`

Restricciones / pistas

- El nombre del programa **no** debe escribirse literal: usá `argv[0]`.
- Si lo ejecutás con otra ruta (`./carpeta/demo`), el mensaje debe reflejarlo.

Criterios de aceptación

- [ ] Muestra `argv[0]` tal cual.
- [ ] El mensaje de uso usa `argv[0]`, no un texto fijo.
- [ ] Cambia correctamente si el programa se ejecuta con otra ruta.

PARTE 4 — Recorrer y clasificar argumentos

4.1 El patrón para recorrer (sin salirse nunca)

Para leer los argumentos que **le pasaste al programa** (no el nombre), se recorre desde `argv[1]` **hasta** `argv[argc-1]`. La condición del bucle es **siempre** `i < argc`.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* Recorro desde 1: el 0 (argv[0]) es el nombre del programa. */
    for (int i = 1; i < argc; i++) {
        printf("Argumento %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

- **Empezamos en 1** porque `argv[0]` es el nombre, no un dato de tu tarea.
- **Terminamos cuando** `i == argc` porque la última posición válida es `argc-1`.

> ***Trampa N.º 1 — salirse del arreglo.*** *Si la condición fuera* `i <= argc`*, en algún momento leerías* `argv[argc]`*, que* ***no existe****: es memoria basura. El resultado es un segfault o datos impredecibles. La condición correcta es* `i < argc`***, sin excepción****.*

4.2 Ahora sí: no todos los argumentos son iguales

En `./programa test.txt --alguna-opcion --dato=2` hay **tres clases** de argumentos. Distinguirlos es el corazón del parseo.

Clase	Ejemplo	Pista para reconocerlo
Posicional	`test.txt`	**No** empieza con `-`. Su **posición** importa (es el "primer dato", el "segundo dato"...).
Opción (flag)	`--alguna-opcion`	Empieza con `-` o `--` y **no** lleva valor. Activa/desactiva algo.
Opción con valor	`--dato=2`	Es una opción **y** trae su valor (con `=` o en el argumento siguiente).

Analogía de pedido por delivery.

- `pedido.txt` → **qué** hay que procesar (posicional).
- `--rapido` → **una opción**: "querés rápido" (sí/no).
- `--cantidad=2` → **opción con valor**: "querés 2".

4.3 La regla del guión

La pista principal para reconocer una opción es simple: **empieza con** `-`.

```
"-"  ...  no es opción (guión solo: suele significar "leer de stdin")
"-x" ...  opción corta
"--" ...  no es una opción (separador especial "fin de opciones")
"--x" ...  opción larga
"x"  ...  argumento posicional
```

Cuidado con los casos de la tabla: un `"-"` solito **no** es una opción, y un `"--"` solito tampoco (suele marcar *"de acá en adelante, todo es posicional"*). A esos casos raros les prestamos atención en la Parte 5.

> ***Trampa N.º 2 — confundir posición con opción.*** *Los **posicionales** dependen del **orden**: el primer nombre de archivo es el de entrada, el segundo el de salida. Las **opciones** suelen poder ir en cualquier orden. No mezcles ambas ideas.*

4.4 Un esqueleto de clasificación (patrón, no solución)

Para clasificar a mano, un patrón típico es mirar la forma del argumento:

```c
for (int i = 1; i < argc; i++) {
    if (argv[i][0] == '-' && argv[i][1] == '-')
        /* opción larga */;
    else if (argv[i][0] == '-')
        /* opción corta */;
    else
        /* posicional */;
}
```

Fijate los índices: `argv[i][0]` es el **primer carácter** de la palabra; `argv[i][1]` el **segundo**. Por eso `argv[i][0] == '-'` pregunta "¿empieza con guión?" y `argv[i][1] == '-'` pregunta "¿y el siguiente también?" (o sea, `--`).

> ***Ojo con esto.*** *Este es un **patrón ilustrativo** para entender cómo se mira una cadena por dentro. Cuidado con los **casos de borde** (`"-"` y `"--"`): el patrón de arriba los trataría como opciones. Un clasificador robusto los contempla aparte.*

Práctica de la Parte 4

Ejercicio 13.5 — Listar solo los parámetros útiles

Conceptos · `argc` , `argv` , bucle, índices

Contexto · Muchas herramientas listan lo que recibieron, sin el nombre del programa. Simplificación: un listado de los argumentos pasados.

Consigna · Escribí un programa que muestre **únicamente** los argumentos posteriores al nombre (desde `argv[1]`), uno por línea, con el formato del escenario de prueba.

Entrada · Los argumentos pasados al ejecutarlo. **Salida** · El listado pedido, sin incluir `argv[0]`.

Escenario de prueba

```
$ ./filtrar -r --debug datos.csv
Parámetros recibidos:
- -r
- --debug
- datos.csv
```

Identificadores sugeridos · `argc`, `argv`, `i`

Restricciones / pistas

- Recorré desde `i = 1`.
- No imprimas `argv[0]`.

Criterios de aceptación

- [ ] No muestra el nombre del programa.
- [ ] Recorre de `1` a `argc-1`.
- [ ] Formato idéntico al ejemplo.

Ejercicio 13.6 — Contador de parámetros extra

Conceptos · `argc`, aritmética simple

Contexto · Los reportes suelen resumir cuántos datos recibieron. Simplificación: contar los argumentos extra.

Consigna · Escribí un programa que informe **cuántos argumentos extra** (sin contar el nombre) recibió, con el formato del escenario de prueba.

Entrada · Los argumentos pasados al ejecutarlo. **Salida** · Una línea con la cantidad.

Escenario de prueba

```
$ ./contar a b c d
Recibiste 4 parámetro(s) extra(s).
```

```
$ ./contar
Recibiste 0 parámetro(s) extra(s).
```

Identificadores sugeridos · `argc` , `extraCount`

Restricciones / pistas

- La cantidad de extra es `argc - 1` . Los corchetes del ejemplo son literales: no hace falta manejar plural/singular finamente.

Criterios de aceptación

- ☐ Con `a b c d` informa `4` .
- ☐ Sin argumentos informa `0` .
- ☐ El cálculo usa `argc` (no cuenta a mano).

Ejercicio 13.7 — Clasificador de argumentos

Conceptos · `argv[i][0]` , comparación de cadenas, clasificación

Contexto · Antes de interpretar opciones, hay que saber qué tipo de argumento es cada uno. Simplificación: clasificar sin ejecutar ninguna acción.

Consigna · Escribí un programa que recorra desde `argv[1]` y clasifique cada argumento como:

- `[OPCION LARGA]` si empieza con `--` y tiene algo después;
- `[OPCION CORTA]` si empieza con **un** `-` y tiene algo después;
- `[POSICIONAL]` si no empieza con `-` .

Entrada · Los argumentos pasados al ejecutarlo. **Salida** · Una línea por argumento con el formato del ejemplo.

Escenario de prueba

```
$ ./clasifica -v --verbose archivo.txt -h
-v -> [OPCION CORTA]
--verbose -> [OPCION LARGA]
archivo.txt -> [POSICIONAL]
-h -> [OPCION CORTA]
```

Identificadores sugeridos · `argc`, `argv`, `i`

Restricciones / pistas

- Distinguí `--` de `-` mirando el **segundo** carácter.
- Contemplá (aunque sea mínimamente) los casos `"-"` y `"--"` para no clasificarlos mal.

Criterios de aceptación

- [ ] Distingue correctamente los tres tipos.
- [ ] No confunde `--` con `-`.
- [ ] Recorre solo los argumentos extra.
- [ ] El formato coincide con el ejemplo.

PARTE 5 — Opciones cortas y largas (`-v` vs `--verbose`)

5.1 Dos estilos, una misma idea

Las herramientas de consola usan **dos estilos** para las opciones. No compiten: **conviven**.

Estilo	Prefijo	Ejemplos	Para qué sirve
Corta	`-` (un guión)	`-v`, `-h`, `-r`, `-o`	Rápida de tipear. Casi siempre letra única.
Larga	`--` (dos guiones)	`--verbose`, `--help`, `--output`	Se lee como palabra. Ideal para no memorizar letras.

Analogía. Es como abreviaturas vs. palabras completas: `tel.` vs. `teléfono`. Las dos sirven; la completa deja menos dudas.

La buena noticia: casi siempre `-v` y `--verbose` **hacen lo mismo**. Es el mismo botón con dos etiquetas. Tu trabajo es que las dos etiquetas activen el mismo comportamiento.

5.2 Opciones cortas apiladas: `-abc`

Un detalle muy cómodo de las opciones **cortas**: se pueden **apilar**. Escribir:

```
./programa -abc
```

es lo mismo que:

```
./programa -a -b -c
```

Son **tres** opciones cortas juntas. Esto **solo** aplica a las cortas (`-`); las largas (`--`) nunca se apilan así.

> ***Ojo con esto.*** *"Apilar" no es gratis: si querés soportarlo* ***a mano****, tenés que recorrer cada carácter del argumento. Por suerte, en la Parte 8 veremos que* `getopt` ***lo hace por vos****.*

5.3 El separador `--` (fin de opciones)

Existe una convención poderosa: un argumento que es **solo** `--` le dice al programa *"de acá en adelante, todo es posicional, aunque empiece con guión"*. Se usa para archivos con nombres raros:

```
./programa -- --archivo-que-empieza-con-guiones.txt
```

Acá `--archivo-...` es un **archivo**, no una opción. Es un caso avanzado; mencionarlo te evita sorpresas cuando leas el código de otras herramientas.

5.4 Detectar una opción con seguridad

Un chequeo básico y seguro (contemplando el caso `"-"`):

```c
if (argv[i][0] == '-' && argv[i][1] != '\0') {
    /* Es una opción: arranca con '-' y no es solo "-" */
}
```

Ese `!= '\0'` pregunta "¿hay algo **después** del guión?". Así un `"-"` solito (que suele significar *"leer de la entrada estándar"*) **no** se confunde con una opción.

Práctica de la Parte 5

Ejercicio 13.8 — Simulador de ayuda

Conceptos · comparación de cadenas (`strcmp`), opciones cortas y largas, terminación del programa

Contexto · Toda herramienta debe poder explicarse sola. `-h` y `--help` son el estándar de facto. Simplificación: mostrar una ayuda fija y salir.

Consigna · Escribí un programa que, si recibe `-h` o `--help` (cualquiera de las dos), muestre un mensaje de ayuda y **termine**. Si no las recibe, no debe mostrar la ayuda.

Entrada · Argumentos por línea de comandos. **Salida** · El mensaje de ayuda (con el nombre del programa incluido), solo si corresponde.

Escenario de prueba

```
$ ./ayuda -h
Uso: ./ayuda [opciones] <archivo>
Opciones:
  -h, --help     Muestra esta ayuda
  -v, --verbose  Modo detallado
```

```
$ ./ayuda --help
Uso: ./ayuda [opciones] <archivo>
Opciones:
```

```
    -h, --help     Muestra esta ayuda
    -v, --verbose  Modo detallado
```

Identificadores sugeridos · `argc`, `argv`, `i`

Restricciones / pistas

- Tenés que reconocer **las dos** variantes: `-h` y `--help`.
- Para comparar cadenas completas usá `strcmp` (no `==`, que compara direcciones).
- Al mostrar ayuda, el programa debe **terminar**: no sigas procesando.
- La ayuda usa `argv[0]` en la línea `Uso: ...`.

Criterios de aceptación

- [ ] Detecta tanto `-h` como `--help`.
- [ ] Son variantes equivalentes (mismo resultado).
- [ ] Tras mostrar la ayuda, el programa **termina**.
- [ ] La línea `Uso:` usa `argv[0]`.

PARTE 6 — Opciones con valor

6.1 El problema

Muchas opciones necesitan **un valor**: `--max=10`, `--output=resultado.txt`, `-o salida.txt`. El valor es el dato que la opción lleva consigo.

Hay **dos formas** de pasarlo, y las dos son válidas:

Forma	Ejemplo	En `argv` queda...
Pegada con `=`	`--dato=2`	**un** argumento: `"--dato=2"`
Separada por espacio	`--dato 2`	**dos** argumentos: `"--dato"` y `"2"`

Analogía. Es como decir *"una Coca-Cola grande"* vs. *"una Coca-Cola, grande"*. Mismo significado, forma distinta.

> ***Momento ajá.*** *Esto es central. Muchos programas aceptan* ***las dos formas****. Cuando parseás a mano, o contemplás ambas, o al menos* ***documentás*** *cuál aceptás.*

6.2 Forma 1: con `=` (todo en un argumento)

Acá el valor vive **dentro** del mismo argumento, después del `=`.

```c
/* ¿el argumento empieza con "--dato="? */
if (strncmp(argv[i], "--dato=", 7) == 0) {
    char *valor = argv[i] + 7;  /* lo que viene después del '=' */
    /* 'valor' apunta a "2" */
}
```

Desarmemos el `+ 7`: `"--dato="` mide 7 caracteres (`-`, `-`, `d`, `a`, `t`, `o`, `=`). Sumar 7 al puntero lo corre justo **hasta el primer carácter del valor**. Eso es **aritmética de punteros**: si `argv[i]` apunta al inicio de la cadena, `argv[i] + 7` apunta al octavo carácter.

> ***Para guardar.*** `strncmp` *compara solo los primeros* `n` *caracteres. Lo usamos para preguntar "¿empieza con* `--dato=` *?" sin importar qué sigue. Después tomamos el resto con* `+ 7`*.*

6.3 Forma 2: separada por espacio (valor en el argumento siguiente)

Acá el valor está en **el siguiente** elemento de `argv`.

```c
/* ¿el argumento es exactamente "--dato" y hay un siguiente? */
if (strcmp(argv[i], "--dato") == 0 && (i + 1) < argc) {
    char *valor = argv[i + 1];  /* el valor es el argumento siguiente */
    i++;                        /* ¡salto el valor para no reprocesarlo! */
    /* usar 'valor' */
}
```

Los dos detalles que importan:

1. `(i + 1) < argc` : comprobar que **existe** un argumento siguiente. Si no, el usuario escribió `--dato` al final y **falta** el valor.
2. `i++` : avanzar un paso extra para que el bucle **no vuelva a procesar** el valor `"2"` como si fuera un argumento independiente.

> ***Trampa N.º 3 — comerse el argumento siguiente.*** *Si te olvidás del* `i++` *, en la próxima vuelta del bucle vas a ver* `"2"` *y lo vas a clasificar como* ***posicional*** *(o peor). El valor de una opción* ***no*** *es un argumento suelto: es parte de la opción. Avanzá un lugar.*

6.4 ¿Y si el valor es un número?

`argv` siempre guarda **texto**. Si la opción representa un número, hay que **convertirlo**:

- `atoi("42")` → `42` (cómodo, pero no avisa si la cadena **no** era un número).
- `strtol("42", &fin, 10)` → más robusto: te deja **verificar** que toda la cadena se convirtió.
- `strtod` para números con decimales.

Elegir bien la conversión es parte de la Parte 7 (robustez).

6.5 Detectar que falta el valor

Si el usuario escribe `--dato` **y no pone** el valor (forma separada), tenés que **detectarlo** y avisar. Ese chequeo es el `(i + 1) < argc` del ejemplo: si es falso, el valor **falta**. Ahí se muestra un error claro y, normalmente, se termina con código de error (Parte 7).

Práctica de la Parte 6

Ejercicio 13.9 — Opción con valor (`--max=N`)

Conceptos · `strncmp` , `atoi` , aritmética de punteros

Contexto · Muchas herramientas aceptan límites configurables. Simplificación: leer un máximo.

Consigna · Escribí un programa que detecte la opción `--max=N` (forma pegada con `=`). Si está presente, muestra `Máximo: N` . Si no está, muestra `Máximo: sin definir` .

Entrada · Argumentos por línea de comandos. **Salida** · La línea del ejemplo.

Escenario de prueba

```
$ ./limite --max=10
Máximo: 10
```

```
$ ./limite
Máximo: sin definir
```

Identificadores sugeridos · `argc`, `argv`, `i`, `maxValue`

Restricciones / pistas

- Detectá el prefijo `--max=` con `strncmp`.
- Extraé el valor sumando al puntero la longitud del prefijo.
- Si no aparece la opción, el valor queda "sin definir" (una bandera booleana te ayuda).

Criterios de aceptación

- [ ] Detecta la forma con `=`.
- [ ] Muestra el valor correcto.
- [ ] Sin la opción, muestra `sin definir`.
- [ ] No crashea si el programa se ejecuta sin argumentos.

Ejercicio 13.10 — Opción con valor separado (`--max 10`)

Conceptos · `strcmp`, índice siguiente, `i++`

Contexto · Otra convención muy común: opción y valor separados por espacio. Simplificación: el mismo máximo, ahora en dos argumentos.

Consigna · Igual que 13.9, pero aceptando la **forma separada por espacio**: `--max 10`.

Entrada · Argumentos por línea de comandos. **Salida** · La misma que 13.9.

Escenario de prueba

```
$ ./limite --max 10
Máximo: 10
```

Identificadores sugeridos · `argc` , `argv` , `i` , `maxValue`

Restricciones / pistas

- Compará la cadena completa con `strcmp` .
- Verificá que exista el argumento siguiente (`i + 1 < argc`).
- No olvides el `i++` para saltar el valor.

Criterios de aceptación

- [ ] Acepta la forma separada por espacio.
- [ ] Salta el valor consumido (`i++`).
- [ ] Verifica que el valor exista antes de usarlo.

Ejercicio 13.11 — Aceptar las dos formas

Conceptos · combinación de `strncmp` y `strcmp` , control de flujo

Contexto · Las herramientas profesionales aceptan múltiples formas de escribir lo mismo. Simplificación: unir la forma pegada y la separada.

Consigna · Mejorá 13.9 y 13.10 en **un solo** programa que acepte **tanto** `--max=10` **como** `--max 10` .

Entrada · Argumentos por línea de comandos. **Salida** · Igual que 13.9.

Escenario de prueba

```
$ ./limite --max=10
Máximo: 10
```

```
$ ./limite --max 10
Máximo: 10
```

Identificadores sugeridos · `argc`, `argv`, `i`, `maxValue`, `haveMax`

Restricciones / pistas

- Podés preguntar primero por el prefijo `--max=` y, si no coincide, por la igualdad exacta `--max`.
- Evitá duplicar la conversión: una sola vez que tengas el texto del valor, convertilo.

Criterios de aceptación

- [ ] Funciona con `=` y con espacio.
- [ ] No hay comportamiento distinto entre las dos formas (mismo resultado).
- [ ] El código no repite innecesariamente la lógica de conversión.

Ejercicio 13.12 — Detectar valor faltante

Conceptos · control de errores, `stderr`, terminación con código

Contexto · Un programa serio no se cuelga ni crashea si al usuario se le olvida el valor. Avisa y termina. Simplificación: error de argumento incompleto.

Consigna · Partiendo de 13.10, si el usuario escribe `--max` **sin valor** (al final de la línea), el programa debe mostrar un error **y terminar**. No debe seguir como si nada.

Entrada · Argumentos por línea de comandos. **Salida** · Mensaje de error y línea de uso.

Escenario de prueba

```
$ ./limite --max
Error: la opción --max necesita un valor
Uso: ./limite [--max=<n> | --max <n>]
```

Identificadores sugeridos · `argc`, `argv`, `i`, `ok`

Restricciones / pistas

- El caso es `i + 1 >= argc` cuando la opción es exactamente `--max`.
- El mensaje de error va por la **salida de errores** (`stderr`), no por la salida normal.
- Terminá el programa con un código **distinto de 0** para señalar el fallo (Parte 7).

Criterios de aceptación

- [ ] Detecta `--max` sin valor.
- [ ] Muestra un error claro (sin crashear).
- [ ] Incluye una línea de uso.
- [ ] Termina con código distinto de 0.

PARTE 7 — Validar, avisar y salir con estilo

7.1 La regla de oro

"Fracasar con claridad es mejor que romperse en silencio."

Un buen programa de consola, ante una entrada inválida:

1. **Avisa** qué pasó, en un mensaje claro;
2. **Muestra** cómo se usa;
3. **Termina** con un código que indique el error.

7.2 Códigos de salida

El `return` de `main` es lo que el **shell** (y cualquier script) ve como **código de salida**:

- `0` → éxito (es el único que significa "todo bien").
- **Distinto de 0** → error (`1` , `2` , etc.).

Por eso, cuando algo falla: `return 1;` . Quien ejecutó tu programa puede **decidir** en función de eso (un `if` en un script, un `&&` en la terminal, una tarea automatizada).

> ***Momento ajá.*** *El código de salida* ***no*** *es un capricho: es la forma en que un programa le* ***cuenta*** *a otro si salió bien. Es la versión "de consola" de un booleano.*

7.3 Errores por `stderr`, resultados por `stdout`

Esto es un detalle **profesional** que marca la diferencia:

- **Resultados normales** → `stdout` (`printf`).
- **Errores, avisos y ayudas** → `stderr` (`fprintf(stderr, ...)`).

¿Por qué? Porque si alguien guarda la salida normal en un archivo:

```
./programa datos.txt > resultado.txt
```

los **errores no deben** colarse en `resultado.txt`. `stdout` va al archivo; `stderr` **sigue apareciendo** en la pantalla, separado. Los datos siguen limpios.

```c
fprintf(stderr, "Error: no se encontró el archivo '%s'\n", nombre);
```

> ***Puente al capítulo siguiente.*** *`stdin`, `stdout` y `stderr` son las **tres tuberías** de todo programa. Acá las usamos apenas; las vas a entender a fondo en `14-entrada-salida-estandar`. Por ahora, quedate con la regla: **errores a `stderr`, resultados a `stdout`**.*

7.4 El mensaje de uso (`--help`)

Toda herramienta que acepta opciones debería tener `-h` / `--help`. Debe ser **conciso, claro** y usar `argv[0]`. Un buen formato de referencia:

```
Uso: ./programa [opciones] <archivo_entrada> [archivo_salida]
Opciones:
  -h, --help           Muestra esta ayuda
  -v, --verbose        Activa modo detallado
  -o, --output <file>  Especifica archivo de salida
  --max <n>            Máximo de elementos (por defecto: 100)
```

La convención de `[]` y `<>` es vieja y muy útil:

- `<>` → **obligatorio** (`<archivo_entrada>`).
- `[]` → **opcional** (`[archivo_salida]` , `[opciones]`).

7.5 Validar argumentos obligatorios

Si tu programa **necesita** un archivo, comprobá que esté:

```c
if (argc < 2) {
    fprintf(stderr, "Error: falta el archivo de entrada\n");
    fprintf(stderr, "Uso: %s <archivo>\n", argv[0]);
    return 1;
}
```

Ese `argc < 2` significa: "solo vino el nombre del programa, no hay ningún dato".

Práctica de la Parte 7

Ejercicio 13.13 — Archivo obligatorio + ayuda

Conceptos · `argc` , `strcmp` , `stderr` , códigos de salida, validación

Contexto · Un procesador de archivos real exige un archivo y ofrece ayuda. Simplificación: la misma estructura, sin procesar nada de verdad.

Consigna · Creá un programa que:

1. Si recibe `-h` o `--help` → muestre la ayuda y retorne `0` .
2. Si **no** recibe ningún argumento extra → muestre un error por `stderr` , la línea de uso, y retorne `1` .
3. Si recibe un nombre de archivo (posicional) → muestre `Procesando: <archivo>` por `stdout` y retorne `0` .

Entrada · Argumentos por línea de comandos. **Salida** · Según el caso, ayuda, error o confirmación.

Escenario de prueba

```
$ ./procesar --help
Uso: ./procesar [opciones] <archivo>
Opciones:
  -h, --help  Muestra esta ayuda
```

```
$ ./procesar
Error: se requiere un archivo
Uso: ./procesar [opciones] <archivo>
```

```
$ ./procesar datos.txt
Procesando: datos.txt
```

Identificadores sugeridos · `argc` , `argv` , `i` , `fileName`

Restricciones / pistas

- La ayuda tiene **prioridad**: detectala **antes** de validar el obligatorio.
- Los **errores** van a `stderr` ; la confirmación normal a `stdout` .
- Usá `argv[0]` en los mensajes de uso.

Criterios de aceptación

- ☐ La ayuda se muestra con `-h` y con `--help` , y retorna `0` .
- ☐ Sin argumentos, el error va a `stderr` y retorna `1` .
- ☐ Con archivo, confirma por `stdout` y retorna `0` .
- ☐ Los mensajes usan `argv[0]` .
- ☐ Los códigos de salida son los correctos.

Ejercicio 13.14 — Validar que el valor sea numérico

Conceptos · `strtol` / `isdigit` , validación, `stderr`

Contexto · `--max=abc` no tiene sentido: el máximo debe ser un número. El programa debe avisarlo. Simplificación: validación de un argumento numérico.

Consigna · Ampliá el programa del 13.11: si `--max` recibe un valor que **no** es un entero válido, mostrá un error y terminá con código de error. Si es válido, mostrá el máximo.

Entrada · Argumentos por línea de comandos. **Salida** · El máximo o un error.

Escenario de prueba

```
$ ./limite --max=abc
Error: '--max' requiere un número entero
```

```
$ ./limite --max=42
Máximo: 42
```

Identificadores sugeridos · `maxText`, `maxValue`, `end`

Restricciones / pistas

- Una forma robusta: `strtol` y comprobar que **consumió toda** la cadena (`*end == '\0'`) y que no hubo desbordamiento.
- El error va a `stderr` y el programa retorna distinto de 0.
- **No** uses `atoi` solo: no distingue `"42"` de `"42abc"` ni de `"abc"`.

Criterios de aceptación

- [ ] `--max=abc` produce un error claro por `stderr`.
- [ ] `--max=42` muestra `Máximo: 42`.
- [ ] Un valor parcialmente numérico (`--max=42x`) se rechaza.
- [ ] El programa no crashea y retorna el código correcto.

PARTE 8 — Orden y robustez: `getopt`

8.1 Por qué existe

Parsear a mano es **perfecto para aprender** (y lo que hiciste hasta acá lo prueba). Pero cuando hay **muchas** opciones, o querés soportar `-abc` apiladas y detectar faltantes de forma consistente, el código a mano se vuelve repetitivo y frágil.

Para eso existe `getopt()`, una función estándar (POSIX) que hace ese trabajo por vos.

> ***Para entender antes de usar. No saltees las Partes 4 a 7.*** *Dominar el parseo manual es lo que te permite entender* ***qué hace*** `getopt` *por dentro, y detectar errores cuando algo no cuadra.*

8.2 Qué resuelve `getopt`

- Maneja opciones **cortas** (`-v` , `-h` , `-o valor`).
- Soporta opciones **apiladas** (`-abc` = `-a -b -c`).
- Sabe cuáles opciones **requieren un valor** y cuáles no.
- Avisa de **opciones desconocidas** y de **valores faltantes**.
- Deja los **posicionales juntos** al final, en `optind` (te lo explico en 8.4).

8.3 La forma de usarlo

```c
#include <unistd.h>  /* getopt */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int opt;

    /* optstring: letras de opciones cortas.
       Un ':' DESPUÉS de una letra significa "esa opción requiere un valor". */
    while ((opt = getopt(argc, argv, "hvf:o:")) != -1) {
        switch (opt) {
        case 'h': /* -h sin valor */        break;
        case 'v': /* -v sin valor */        break;
        case 'f': /* -f <archivo> */        break;
        case 'o': /* -o <archivo> */        break;
        case '?': /* opción desconocida o valor faltante */ break;
        default:                            break;
        }
    }
    return 0;
}
```

Desarmemos el `"hvf:o:"`:

Letra en `optstring`	Significa
`h`	opción `-h` (sin valor)
`v`	opción `-v` (sin valor)
`f:`	opción `-f` que **requiere** un valor (el `:` va **después** de la letra)
`o:`	opción `-o` que **requiere** un valor

El valor de una opción con `:` queda en la variable global **`optarg`** (`char *`).

> ***Momento ajá.*** *El* `:` *va* ***a la derecha*** *de la letra que necesita argumento. Leer* `"f:"` *es decir "la opción* `-f` *lleva un valor". Leer* `"hf:"` *es decir "* `-h` *sin valor,* `-f` *con valor".*

8.4 Los posicionales después de `getopt`

Cuando `getopt` termina (devuelve `-1`), todas las opciones ya se procesaron. La variable global `optind` apunta al **primer argumento que no es opción**. De ahí en adelante, todo es posicional:

```c
while ((opt = getopt(argc, argv, "hv")) != -1) { /* ... */ }

for (int i = optind; i < argc; i++) {
    printf("Posicional: %s\n", argv[i]);
}
```

`getopt` **reordena** internamente la línea para dejarte las opciones primero y los posicionales después. Vos solo leés desde `optind`.

8.5 Errores que detecta

- **Opción desconocida:** `getopt` devuelve `'?'` (y suele imprimir un aviso por `stderr`).
- **Valor faltante:** si una opción con `:` aparece sin su valor, también se señala con `'?'` (o `:` según configuración). Lo habitual es tratar `'?'` mostrando tu propio mensaje de uso, más claro para el lector.

8.6 Disponibilidad

`getopt` viene de `<unistd.h>`, típico de Linux y macOS. En Windows, con MinGW/GCC o entornos POSIX (como Git Bash / MSYS2), suele estar disponible. Si compilás con MSVC puro, no; ahí este capítulo te sirve igual porque ya sabés hacerlo a mano.

Práctica de la Parte 8

Ejercicio 13.15 — `getopt` básico

Conceptos · `getopt`, `optstring`, `switch`, opciones apiladas

Contexto · Reescribir el parseo con la función estándar. Simplificación: tres opciones cortas.

Consigna · Usando `getopt`, interpretá:

- `-h` → mostrá la ayuda y terminá;
- `-v` → activá modo detallado (informá `Modo verbose: ON`);
- `-f <archivo>` → requiere un archivo (`Archivo: <archivo>`).

Entrada · Argumentos por línea de comandos. **Salida** · Según las opciones recibidas.

Escenario de prueba

```
$ ./prog -v -f datos.txt
Modo verbose: ON
Archivo: datos.txt
```

```
$ ./prog -vf datos.txt
Modo verbose: ON
Archivo: datos.txt
```

Identificadores sugeridos · `opt`, `optstring`, `optarg`

Restricciones / pistas

- El `optstring` correcto para esto es `"hvf:"`.
- El segundo escenario prueba que `-vf` **apiladas** funciona sin código extra (lo hace `getopt`).
- Para `-h`, terminá el programa dentro del `case`.

Criterios de aceptación

- [ ] Usa `getopt` con `optstring` correcto.
- [ ] Acepta `-v` y `-f` juntas (`-vf`).
- [ ] `-f` toma su valor de `optarg`.
- [ ] `-h` muestra ayuda y termina.

Ejercicio 13.16 — Posicionales con `getopt`

Conceptos · `optind`, posicionales, orden de argumentos

Contexto · Separar las opciones de los datos "de verdad". Simplificación: listar los archivos sueltos después de las opciones.

Consigna · Ampliá 13.15 para que, **después** de procesar las opciones, muestre todos los posicionales restantes (los que quedan desde `optind`).

Entrada · Mezcla de opciones y posicionales. **Salida** · El efecto de las opciones y luego la lista de posicionales.

Escenario de prueba

```
$ ./prog -v entrada.txt salida.txt extra.txt
Modo verbose: ON
Posicionales:
  entrada.txt
  salida.txt
  extra.txt
```

Identificadores sugeridos · `opt`, `optind`, `i`

Restricciones / pistas

- Recorré desde `optind` hasta `argc-1`.
- No mezcles: las opciones se procesan en el `while`; los posicionales, **después**.

Criterios de aceptación

- [ ] Usa `optind` para ubicar el primer posicional.
- [ ] Lista todos los posicionales en orden.
- [ ] Las opciones no se mezclan con la lista.

PARTE 9 — Nivel pro: `getopt_long`

9.1 El salto

Ahora sí, **nivel pro**: opciones **largas** (`--verbose` , `--help` , `--output=...` , `--max 10`).

Para eso existe `getopt_long()` (también POSIX, header `<getopt.h>`). Es **el estándar** que usan herramientas como `ls` , `grep` y `gcc` . Todo lo que aprendiste de `getopt` sigue valiendo: `getopt_long` es su hermano mayor, que además entiende las palabras largas.

9.2 Comparación rápida

Característica	`getopt`	`getopt_long`
Opciones cortas (`-v`)	Sí	Sí
Opciones largas (`--verbose`)	**No**	**Sí**
Acepta `--opcion=valor`	No	**Sí**
Acepta `--opcion valor`	No	**Sí**
Manejo de `-abc` apiladas	Sí	Sí

9.3 Las tres piezas

Para usar `getopt_long` necesitás **tres** cosas:

1. incluir `<getopt.h>` ;
2. declarar una **tabla** de opciones largas (`struct option`);
3. llamar a `getopt_long` en lugar de `getopt` .

9.4 La tabla de opciones largas

```c
#include <getopt.h>

static struct option long_options[] = {
    {"help",    no_argument,       0, 'h'},
    {"verbose", no_argument,       0, 'v'},
    {"output",  required_argument, 0, 'o'},
    {"max",     required_argument, 0, 'm'},
    {0, 0, 0, 0}   /* terminador obligatorio */
};
```

Cada fila tiene **cuatro columnas**:

1. **Nombre largo** (`"help"`) → se reconoce como `--help` .
2. **¿Lleva valor?** → `no_argument` (flag) o `required_argument` (con valor).
3. **¿A dónde apunta el resultado?** → `0` significa "devolvé el valor de la cuarta columna".
4. **Código corto asociado** → `'h'` , `'v'` , `'o'` , `'m'` . Así la opción larga **y** la corta caen en el **mismo** `case` del `switch` .

La última fila `{0,0,0,0}` es el **terminador**: le dice a `getopt_long` dónde acaba la tabla. **No la olvides** (es el bug clásico de esta función).

> ***Momento ajá.*** *Gracias a la cuarta columna,* `--verbose` *y* `-v` *comparten el* `case 'v'` *. Escribís el comportamiento* ***una sola vez*** *y las dos formas lo activan. Eso es elegancia, no magia.*

9.5 Cómo se llama

```
int opt;
int option_index = 0;

while ((opt = getopt_long(argc, argv, "hvo:m:",
                          long_options, &option_index)) != -1) {
    switch (opt) {
    case 'h': /* -h o --help */           break;
    case 'v': /* -v o --verbose */        break;
    case 'o': /* -o o --output (optarg) */break;
    case 'm': /* -m o --max (optarg) */   break;
    case '?': /* desconocida / falta valor */ break;
    default:                              break;
    }
}
```

El primer string (`"hvo:m:"`) sigue siendo el `optstring` de las opciones **cortas**. La tabla es para las **largas**. Y `getopt_long` entiende, sin costo extra:

- `--output=archivo.txt` (pegado con `=`);
- `--output archivo.txt` (separado por espacio).

Para las largas, el valor llega —igual que siempre— en `optarg` .

9.6 Errores

- Opción larga **desconocida** → `'?'` ;
- Opción larga que **requiere valor** y no lo tiene → `'?'` ;
- Opción corta **desconocida** → `'?'` .

Como en `getopt` , lo más limpio es manejar `'?'` mostrando **tu** mensaje de uso y retornando el código de error.

Práctica de la Parte 9

Ejercicio 13.17 — (JEFE) Utilidad de consola completa

Conceptos · `getopt_long`, `struct option`, `optarg`, `optind`, `stderr`, códigos de salida, validación numérica, ayuda.

Contexto · Reunir **todo** el capítulo en una sola herramienta real de línea de comandos. Es una **simplificación**: no procesa archivos de verdad, pero su interfaz es la de una utilidad profesional.

Consigna · Desarrollá un programa que interprete estas opciones:

Corta	Larga	¿Valor?	Acción
`-h`	`--help`	no	Muestra la ayuda y termina con `0`.
`-v`	`--verbose`	no	Activa el modo detallado.
`-o`	`--output`	sí	Guarda el archivo de salida indicado.
`-m`	`--max`	sí	Guarda el máximo (número entero).

Y que además:

1. Acepte las **dos formas** de valor para las largas (`--output=x` y `--output x`).
2. Trate los argumentos **no-opción** como archivos posicionales (listados al final).
3. Valide que `--max` sea un entero; si no, error por `stderr` y retorno distinto de 0.
4. Ante una opción desconocida, muestre un error por `stderr`, la línea de uso y retorne distinto de 0.
5. Si no recibe ningún posicional **ni** `--output`, avise que no hay nada que hacer (error claro).

Entrada · Argumentos por línea de comandos. **Salida** · El efecto de las opciones por `stdout`; los errores por `stderr`.

Escenario de prueba

```
$ ./util --verbose --max=10 entrada.txt salida.txt
Modo verbose: ON
Máximo: 10
Posicionales:
  entrada.txt
  salida.txt
```

```
$ ./util --max
Error: la opción --max necesita un valor
Uso: ./util [opciones] <archivo>...
```

```
$ ./util --max=abc
Error: '--max' requiere un número entero
```

```
$ ./util --desconocida
Error: opción desconocida
Uso: ./util [opciones] <archivo>...
```

Identificadores sugeridos · `opt` , `long_options` , `option_index` , `optind` , `verbose` , `max_value` , `output_name` , `have_output` , `i`

Restricciones / pistas

- Usá `getopt_long` con el `optstring` `"hvo:m:"` (o el que corresponda a tus short options).
- **No** olvides el terminador `{0,0,0,0}` de la tabla.
- El `-h` tiene **prioridad**: mostralo apenas aparece y terminá.
- Validá `--max` con `strtol` comprobando que se consumió **toda** la cadena.
- Errores → `stderr` ; resultados → `stdout` .
- Los posicionales salen de `argv[optind..argc-1]` .

Criterios de aceptación

- [ ] Reconoce `-v` / `--verbose`, `-o` / `--output` y `-m` / `--max`.
- [ ] Acepta valor pegado (`--max=10`) y separado (`--max 10`).
- [ ] `--help` / `-h` muestra ayuda y retorna `0`.
- [ ] `--max` sin valor y `--max=abc` producen errores claros por `stderr` con código de retorno != 0.
- [ ] Una opción desconocida produce error por `stderr` y código != 0.
- [ ] Los posicionales se listan después de las opciones (usando `optind`).
- [ ] Los resultados van a `stdout` y los errores a `stderr`.
- [ ] No hay accesos fuera de rango ni bucles infinitos.

Cierre: lo que te llevás

Si tuvieras que quedarte con **una sola idea**, que sea esta: **tu programa puede recibir sus órdenes al nacer**, y esas órdenes llegan como una lista de cadenas que se llama `argv`.

- **`argc` cuenta, `argv` guarda.** `argc` es cuántos argumentos hay (incluido el nombre); `argv` es la lista, con `argv[0]` = el nombre del programa.
- **El espacio separa; las comillas agrupan.** `"hola mundo"` entre comillas es **un** argumento.
- **Recorré con `i < argc`.** Nunca toques `argv[argc]` : es la trampa N.º 1.
- **Aprendé a clasificar:** posicional (sin `-`), opción corta (`-x`), opción larga (`--xx`), y opción con valor (`--xx=valor` o `--xx valor`).
- **Ojo con la trampa N.º 3:** cuando el valor viene separado por espacio, avanzá un índice (`i++`) para no reprocesarlo.
- **Fracasá con claridad:** errores por `stderr` , ayuda con `argv[0]` , y `return` distinto de 0 ante error.
- **Dominá lo manual primero** (Partes 4–7); recién después **delegate** en `getopt` y `getopt_long` , que te dan orden, opciones apiladas y soporte de `--largas` con valor.

Y el puente hacia lo que viene: hasta ahora viste **cómo entra la orden** (por la línea de comandos). En `14-entrada-salida-estandar` vas a aprender **cómo entra y sale todo lo demás**: las tres tuberías `stdin` / `stdout` / `stderr` , la lectura por teclado, los búferes y `EOF` . Los

argumentos te dijeron qué hacer; el capítulo siguiente te enseña a **conversar** con el programa mientras corre.