Capítulo 13 — `getopt_long`: el analizador de opciones, explicado bien

> *Capítulo intermedio, pero **escrito para que se entienda de verdad**. Se apoya en variables (`02`), operadores (`03`), `if` y bucles (`04`), funciones (`05`) y cadenas (`06`).*
>
> *Es el capítulo más práctico del libro: aprendés **una sola función** (`getopt_long`) y la domas hasta que no te encuentres nunca con una herramienta de consola que no sepas interpretar. En el siguiente capítulo (`14-entrada-salida-estandar`) vas a profundizar en las tres tuberías (`stdin` / `stdout` / `stderr`), que acá solo necesitás para mostrar los resultados.*

Cómo leer este capítulo

Hasta ahora, todos tus programas **preguntan** mientras corren: te muestran un cartel y esperan a que escribas algo por teclado. Las herramientas serias -- `ls`, `grep`, `git`, `gcc` -- no hacen eso: les pasás las órdenes cuando las ejecutás.

Este capítulo te enseña **una sola función** para hacerlo: `getopt_long()`. No tres funciones, no diez. Una. Pero la vas a entender tan bien que podrás leer el código de cualquier herramienta y saber exactamente qué hace cada cosa.

Está dividido en **ocho partes**. Después de cada concepto, una **práctica** --uno o más ejercicios cortos-- para que lo uses ya mismo. No saltees las prácticas: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	El problema de parsear a mano	Escribir un parser de opciones desde cero es más trabajo del que parece.
2	La llamada a `getopt_long`	La firma completa, parámetro por parámetro.
3	Los tres macros	`no_argument`, `required_argument` y `optional_argument`.
4	El struct de opciones largas	`struct option`: la tabla que conecta nombres largos con códigos cortos.
5	Las variables globales	`optarg`, `optind`, `optopt`: lo que `getopt_long` deja atrás para que las uses.
6	Errores y el caso `'?'`	Qué pasa cuando algo sale mal y cómo manejarlo con estilo.
7	Posicionales después de `optind`	Cómo recuperar los argumentos "de verdad" una vez terminada la magia.
7.5	La ayuda de uso	Un formato estándar para guiar al usuario (sin función en C, pero con un helper).
8	El ejercicio jefe	Juntás todo en una herramienta de consola completa.

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 -- El problema de parsear a mano

1.1 El parser manual: tu primer intento (y por qué dejarlo)

Imaginate que querés hacer un programa al que se le puedan pasar opciones como estas:

```
./miapp --help -v --max=42 archivo.txt
```

Tu primer impulso es escribir algo así:

```
for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "--help") == 0) { /* ... */ }
    else if (strcmp(argv[i], "-v") == 0) { /* ... */ }
    else if (strncmp(argv[i], "--max=", 6) == 0) { /* ... */ }
}
```

Esto funciona para tres opciones. Pero mirá lo que viene:

- ¿Qué pasa si `-v` puede ir con valor (`-v 5`) **y** sin él (`-v`)?
- ¿Cómo hacés para soportar opciones apiladas (`-abc` = `-a -b -c`)?
- ¿Y si el usuario escribe `--max=abc` , `--MAX` , `--max ` , o simplemente `--max` al final?
- ¿Dónde separás las opciones de los archivos posicionales?

Cada opción nueva duplica tu código. Cada caso de error lo escribís a mano. Es un trabajo de hormiga que **ya alguien hizo por vos**.

> ***Para guardar.*** *Esto no es una crítica a tu capacidad: es un reconocimiento a que* `getopt_long` *fue escrito por personas que saben que repetimos el mismo patrón en cada programa.* ***Delegatemos lo aburrido.***

1.2 La alternativa: delegar en `getopt_long`

`getopt_long()` es una función estándar POSIX que recibe los argumentos de `main`, busca las opciones que vos le pediste, y te devuelve **una por una** hasta agotarlas. Vos solo escribís un `switch` con lo que querés hacer con cada opción; el resto lo resuelve la función:

- opciones apiladas (`-abc`)
- valores pegados (`--max=42`) y separados (`--max 42`)
- detección de opciones desconocidas
- aviso de valores faltantes
- reordenamiento interno (deja los posicionales juntos al final)

Todo eso, en **una sola llamada**. La contraparte corta (`getopt`) solo entiende `-x`. `getopt_long` entiende **tanto** `-x` como `--opcion-larga`. Eso es todo lo que importa para este capítulo.

Práctica de la Parte 1

Ejercicio 13.1 -- Contar líneas de código manuales

Conceptos · parseo manual, comparación de cadenas, complejidad

Contexto · Antes de ver la solución, necesitás sentir **cuánto trabajo** evita `getopt_long`. Simplificación: no escribís nada. Solo calculás.

Consigna · Dado este conjunto de opciones, ¿cuántas comparaciones individuales (`strcmp` o `strncmp`) tendrías que escribir para reconocerlas **todas** con el método manual?

Opción	Requiere valor?	Formas aceptadas
`-h` / `--help`	no	`-h` , `--help`
`-v` / `--verbose`	no	`-v` , `--verbose`
`-o` / `--output`	sí	`-o f` , `-of` , `--output=f` , `--output f`
`-m` / `--max`	sí	`-m n` , `-mn` , `--max=n` , `--max n`

Contá cada comparación individual que tendrías que escribir en tu `if/else`. No uses `getopt_long`. Solo pensá.

Entrada · Ninguna (ejercicio de análisis). **Salida** · El número total de comparaciones y una lista breve de por qué son necesarias.

Criterios de aceptación

- [ ] Contaste cada forma individual, no cada opción.
- [ ] Consideraste pegado con `=` y separado por espacio como formas distintas.
- [ ] Reconociste que `-of` necesita recorrer carácter por carácter.

PARTE 2 -- La llamada a `getopt_long`

2.1 La firma completa

Acá está la función. Memorizá la forma; después desarmamos cada pieza.

```
int getopt_long(int argc,
                char *const argv[],
                const char *optstring,
                const struct option *longopts,
                int *longindex);
```

Pero en la práctica casi siempre la llamás así:

```
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 */ break;
        case 'm': /* -m o --max */ break;
        case '?': /* error */ break;
    }
}
```

Esa es **toda** la estructura. Una sola vez declarás las opciones (en `long_options` y en el `optstring`). Después, el bucle se repite automáticamente hasta que devuelve `-1`, y en cada vuelta te dice cuál opción encontró a través de `opt`.

2.2 Desglose parámetro por par

Parámetro	Qué es	Qué le pasás	Por qué importa
`argc`	Cantidad de argumentos	El mismo `argc` de `main`	Necesita la lista cruda
`argv`	Arreglo de cadenas	El mismo `argv` de `main`	Recorre los argumentos por vos
`optstring`	Opciones cortas (ver Parte 3)	`"hvo:m:"`	Dice cuáles son cortas y cuáles llevan valor
`longopts`	Tabla de opciones largas (ver Parte 4)	Nombre de tu array de `struct option`	Dice qué nombres largos reconoce
`longindex`	Puntero a int (opcional, podés poner `NULL`)	`&option_index`	Si lo pasás, te dice en qué posición de la tabla encontró la opción

El **retorno** de cada llamada:

- Un `char` convertido a `int`: el código de la opción encontrada (`'h'`, `'v'`, `'o'`, `'m'`).
- `'?'` si la opción es desconocida o le falta un valor requerido.
- `-1` cuando ya no quedan opciones: el bucle se rompe.

> ***Ojo con esto.*** *El retorno es `int` (no `char`) porque necesitás representar `-1` junto con los códigos de opción. Si lo hubieran devuelto como `char`, en muchos sistemas `-1` y `'ÿ'` sería el mismo valor, y no podrías distinguirlos. Por eso la variable se llama `opt` (un `int`) y* ***nunca*** *`char opt`.*

2.3 El patrón que se repite siempre

Cada programa que use `getopt_long` sigue **exactamente el mismo esqueleto**:

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

int main(int argc, char *argv[])
{
    int opt;
    int option_index = 0;

    /* Declará las opciones largas ANTES del bucle. */
    static struct option long_options[] = { ... };

    /* Bucle: getopt_long devuelve la opción en cada vuelta. */
    while ((opt = getopt_long(argc, argv, "hvo:m:",
                              long_options, &option_index)) != -1) {
        switch (opt) {
            case 'h': break;  /* una opción */
            case 'v': break;  /* otra opción */
            case '?': break;  /* error */
        }
    }

    return 0;
}
```

Fijate que **el bucle y el switch son siempre iguales**. Lo único que cambia entre un programa y otro es:

1. El contenido de `long_options` (qué opciones largas reconocés).
2. El string de opciones cortas (el tercer argumento de `getopt_long`).
3. Los cuerpos de los `case`.

> ***Momento ajá.*** *Una vez que dominás este esqueleto, podés copiarlo a cualquier proyecto y solo cambiar las opciones. Es como un molde: la forma no cambia, solo el contenido.*

Práctica de la Parte 2

Ejercicio 13.2 -- Leer la firma sin compilar

Conceptos · parámetros de `getopt_long`, tipos

Contexto · Antes de correr código, necesitás saber **qué recibe** y **qué devuelve**. Simplificación: interpretar una llamada a `getopt_long`.

Consigna · Dada esta llamada:

```
while ((opt = getopt_long(argc, argv, "hvfo:", my_options, NULL)) != -1) { ... }
```

respondé sin compilar:

1. ¿Cuántas opciones **cortas** se están declarando? ¿Cuáles son?
2. ¿Cuáles llevan valor (y cuáles no)? Mirá el `optstring`.
3. ¿Se están usando las opciones largas? ¿Cómo lo sabés?
4. ¿Qué pasa si le pasás `NULL` como último parámetro (`longindex`)? ¿Ocurre algo malo?

Entrada · La línea de llamada mostrada arriba. **Salida** · Las cuatro respuestas.

Criterios de aceptación

- [ ] Identificás las opciones cortas en el string.
- [ ] Distinguís cuáles necesitan valor por el `:` siguiente.
- [ ] Sabés que `NULL` en `longindex` es válido: simplemente no se llena la información del índice.
- [ ] El `my_options` es un array de `struct option`, aunque no lo veas declarado acá.

Ejercicio 13.3 -- Completar el esqueleto

Conceptos · estructura básica, variables globales necesarias

Contexto · Vos tenés que escribir las variables que faltan para que el esqueleto compile (aunque no haga nada). Simplificación: declarar lo mínimo para compilar un programa con `getopt_long`.

Consigna · Escribí la parte principal de un programa C (desde `main` hasta el cierre) que llame a `getopt_long` con estas opciones:

- Cortas: `-h` , `-v` , `-d <archivo>`
- Largas: `--help` , `--verbose` , `--debug-file` (esta última requiere un valor como archivo)

El cuerpo de cada `case` puede estar vacío. Lo importante es que las **declaraciones** sean correctas.

Criterios de aceptación

- ☐ Se incluye `<getopt.h>` .
- ☐ El `optstring` corto es `"hvd:"` .
- ☐ El array de `struct option` tiene filas para `help` , `verbose` y `debug-file` .
- ☐ Cada fila larga declara correctamente si lleva o no argumento.
- ☐ El bucle es `while ((opt = getopt_long(...)) != -1)` .

PARTE 3 -- Los tres macros que definen "si lleva valor"

3.1 ¿Qué significa "argumento" en este contexto?

Antes de leer la tabla, una aclaración importante: cuando `getopt_long` habla de **argumento**, no se refiere a los argumentos de `main` (`argc` / `argv`). Se refiere a **si la opción lleva consigo un valor adicional**.

- `-v` → **no** lleva nada. Es un interruptor: prende o apaga algo.
- `-d archivo.txt` → **sí** lleva un valor adicional: `"archivo.txt"` .

Ese "sí" o "no" es exactamente lo que dicen los macros de esta parte.

3.2 Los tres valores posibles

Cada fila de tu tabla de `struct option` necesita una respuesta a esta pregunta: **¿la opción larga lleva valor?** Las opciones son tres:

Macro	Significa	Ejemplo	¿Se usa para?
`no_argument`	La opción **no** lleva valor (es un flag)	`--help`, `-v`	Interruptores booleanos
`required_argument`	La opción **sí** lleva valor	`--output=archivo`, `-d archivo`	Valores obligatorios
`optional_argument`	La opción **puede** llevar valor, pero no es obligatorio	`--mode=fast` (ó simplemente `--mode`)	Valores opcionales con sentido por defecto

Solo estos tres. No hay un cuarto. Si una opción lleva algo adicional, es uno de los dos primeros. Si puede ir sin nada, es el tercero.

Analogía. Pensá en las opciones como **interruptores de luz**:

- `no_argument` → un interruptor normal: prendé/apagá, punto.
- `required_argument` → un dimmer (regulador): necesitás decir **hasta dónde** querés bajar la luz.
- `optional_argument` → un dimmer con preset: si lo tocás solo prende al nivel medio; si lo girás, elegís el nivel vos.

3.3 El truco de los nombres

Los macros se llaman `no_argument`, `required_argument` y `optional_argument`. Notá que **argument** lleva una "u" --es del inglés. En el contexto de esta función, "argument" = "valor adicional de la opción". Nunca te confundas con `argc` / `argv`: son cosas distintas.

> ***Para guardar.*** *Cuando veas* `required_argument`*, leelo como* ***"esta opción lleva un valor"*** *--no como "requiere un argumento de función", que es otra cosa totalmente.*

Práctica de la Parte 3

Ejercicio 13.4 -- Clasificar opciones por su macro

Conceptos · `no_argument`, `required_argument`, `optional_argument`

Contexto · Antes de escribir código, necesitás **leer** una lista de opciones y saber qué macro corresponde a cada una. Simplificación: solo elegís el macro correcto.

Consigna · Para cada opción, escribí qué macro corresponde en la tabla de `struct option`:

Opción	¿Lleva valor adicional?	Macro correcto
`--help`	no	?
`--max 10`	sí, el número	?
`--output=fichero.txt`	sí, el nombre de archivo	?
`--verbose` (solo activa flag)	no	?
`--color` (si acepta `--color=auto` ó solo `--color`)	puede tener o no	?

Entrada · Las opciones listadas arriba. **Salida** · La tabla completada con los macros correctos.

Criterios de aceptación

- [ ] `no_argument` para las que son puro interruptor (flag).
- [ ] `required_argument` para las que **siempre** llevan un valor.
- [ ] `optional_argument` para la que puede ir con o sin valor.

PARTE 4 -- El struct de opciones largas

4.1 ¿Qué es `struct option`?

Cada fila de tu tabla es una estructura de cuatro campos:

```
struct option {
    const char *name;          /* el nombre SIN los guiones */
    int         has_arg;       /* no_argument, required_argument u optional_argument */
    int        *flag;           /* NULL si querés comportamiento estándar */
    int         val;           /* código corto que aparece en el switch */
};
```

Y lo usás así:

```
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 */
};
```

La **última fila es obligatoria**: `{0, 0, 0, 0}`. Sin ella, `getopt_long` no sabe dónde termina tu tabla y va a leer memoria de más --potencialmente un crash o comportamiento impredecible.

4.2 Desglose de cada campo

Campo 1: name -- el nombre largo sin los guiones

```
{"help", ...}       /* corresponde a --help */
{"verbose", ...}    /* corresponde a --verbose */
```

Acá escribís **solo la palabra**, sin los guiones. `getopt_long` sabe que si la tabla contiene `"help"`, eso significa `--help`. Nunca ponés los guiones en este campo.

Campo 2: has_arg -- ¿lleva valor?

Usás uno de los tres macros de la Parte 3:

```
{"help", no_argument, 0, 'h'},          /* --help --> flag, sin valor */
{"max", required_argument, 0, 'm'},     /* --max=10 o --max 10 --> lleva un valor */
```

Campo 3: flag -- lo que va a pasar con el retorno (casi siempre 0)

Este es el campo que más confusión genera. Tiene dos modos:

Modo A (el que vas a usar el 99% de las veces): `flag = 0`. En este modo, `getopt_long` devuelve en cada vuelta el valor del cuarto campo (`val`). Así, `--help` devuelve `'h'`, que coincide con `-h`, y ambos van al mismo `case 'h':`.

```
{"help", no_argument, 0, 'h'},   /* devuelve 'h' (como -h) */
```

Modo B: `flag` apunta a una variable. En este caso, `getopt_long` **no** usa el cuarto campo. En vez de devolver un código, escribe `1` en la variable apuntada por `flag` y devuelve `0`. Es útil para opciones largas que no tienen equivalente corto, pero es un patrón avanzado que no vamos a usar acá.

Para este capítulo, **siempre poné `0`**.

Campo 4: val -- el código que aparece en tu switch

Este es el valor de retorno cuando se encuentra la opción larga. Lo más común es que sea igual al código de la opción corta correspondiente:

```
{"help", no_argument, 0, 'h'},    /* --help devuelve 'h' */
{"verbose", no_argument, 0, 'v'}, /* --verbose devuelve 'v' */
```

Así, `--help` y `-h` caen en el **mismo** `case 'h':`. Escribís la lógica una vez y sirve para los dos.

> ***Para guardar.*** *El cuarto campo es el* ***puente****: une la opción larga con el código corto del* `switch`*. Si pones* `'h'`*, ambas variantes (* `-h` *y* `--help` *) van al mismo* `case`*.*

4.3 El terminador: {0, 0, 0, 0}

Esta fila dice a `getopt_long`: "acá terminó mi tabla". Es un arreglo, no una lista con un conteo adjunto --la única forma de saber dónde termina es encontrar ceros. Si la olvidás:

- `getopt_long` lee memoria más allá del array → valores basura.
- Probablemente termine en un crash o en comportamientos imposibles de diagnosticar.
- El error **no se detecta** en tiempo de compilación: aparece solo al ejecutar (si tenés suerte).

No la olvides. Nunca. La ponés siempre, sin excepciones, incluso si tu tabla tiene una sola opción.

4.4 Uniendo las partes: un ejemplo completo de tabla

```
static struct option long_options[] = {
    {"help",            no_argument,       0, 'h'},
    {"verbose",         no_argument,       0, 'v'},
    {"debug",           optional_argument, 0, 'd'},
    {"output",          required_argument, 0, 'o'},
    {"max-retries",     required_argument, 0, 'm'},
    {0, 0, 0, 0}
};
```

Notá que `"max-retries"` tiene guión medio (`-`). Eso es normal: el nombre de la opción larga puede llevar cualquier carácter que no sea `-`. Y sí, `getopt_long` entiende exactamente esa palabra como `--max-retries`.

> ***Ojo con esto.*** *La tabla solo define* ***opciones largas****. Para las opciones cortas (* `-h`*,* `-v`*), necesitás también el* `optstring` *(el string del tercer parámetro de* `getopt_long`*). Si ponés una opción larga en la tabla pero no la incluís en el* `optstring`*, igual funciona. Pero si ponés una corta en el* `optstring` *y no en la tabla, solo funciona como corta (obvio).*

Práctica de la Parte 4

Ejercicio 13.5 -- Escribir la tabla de opciones

Conceptos · `struct option`, los cuatro campos, terminador

Contexto · La tabla de opciones es el corazón de `getopt_long`: sin ella, no hay opciones largas. Simplificación: escribir una tabla correcta para un conjunto dado de opciones.

Consigna · Escribí la declaración completa de `long_options` (array de `struct option`) para un programa que acepta:

Opción corta	Opción larga	¿Lleva valor?
`-h`	`--help`	no
`-v`	`--verbose`	no
`-c`	`--config-file`	sí (nombre de archivo)
`-t`	`--timeout`	sí (número en segundos)

No escribís la llamada a `getopt_long` ni el bucle. Solo la tabla.

Entrada · La tabla de opciones arriba. **Salida** · El código C completo para declarar el array.

Identificadores sugeridos · `long_options`

Criterios de aceptación

- [ ] Cada fila tiene los cuatro campos en orden.
- [ ] `name` no incluye los guiones (`"help"`, no `"--help"`).
- [ ] El segundo campo usa el macro correcto para cada opción.
- [ ] La última fila es `{0, 0, 0, 0}`.
- [ ] Los valores del cuarto campo coinciden con las letras de las opciones cortas.

Ejercicio 13.6 -- Debuggear una tabla rota

Conceptos · errores comunes en `struct option`

Contexto · Las tablas mal escritas son la fuente más común de bugs con `getopt_long`. Simplificación: encontrar los errores en una tabla dada.

Consigna · Esta tabla tiene **tres errores**. Encontralos y escribí qué está mal en cada fila:

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

Entrada · La tabla con errores arriba. **Salida** · Tres errores identificados, uno por fila (la última fila es correcta).

Criterios de aceptación

- [ ] Identificás el `has_arg` incorrecto en `help`.
- [ ] Encontrás que `"v"` es una cadena, no un char.
- [ ] Reparás el `flag = NULL` cuando debería ser `0`.
- [ ] Reconocés que la fila de `max` está correcta (es un distractor).

PARTE 5 -- Las variables globales: lo que `getopt_long` deja atrás

5.1 La analogía del cajero automático

Cuando vas a un cajero, le pasás tus datos y él **guarda información internamente** para el siguiente paso: tu número de cuenta, la fecha límite, etc. Vos no necesitás recordarlo --el cajero lo tiene en mente y te lo va preguntando.

`getopt_long` hace exactamente eso con tres **variables globales**: las modifica para que vos puedas consultarlas cuando necesités un dato adicional. Estas variables están declaradas en `<getopt.h>`; no las declarás vos, simplemente las usás.

5.2 optarg: el valor de la opción

Cada vez que `getopt_long` encuentra una opción que **requiere un valor** (`-o archivo.txt` , `--output archivo.txt` o `--output=archivo.txt`), guarda ese valor en la variable global `optarg` .

```
while ((opt = getopt_long(argc, argv, "o:m:", ...)) != -1) {
    switch (opt) {
        case 'o':
            printf("Archivo de salida: %s\n", optarg);  /* <-- aca esta! */
            break;
        case 'm':
            printf("Valor maximo: %s\n", optarg);         /* <-- tambien aca */
            break;
    }
}
```

`optarg` es un `char *` . Puntero a cadena. Siempre contiene **lo que viene después** de la opción, ya sea pegado con `=` o separado por espacio. No necesitás hacer aritmética de punteros: `getopt_long` ya lo resolvió.

> ***Momento ajá.*** *Esto es exactamente lo que resolvíamos a mano en el capítulo anterior: sumar un offset al puntero para encontrar el valor después del* `=` *. Acá, la función lo hace por vos y te lo deja en una variable lista para usar.*

5.3 optind: donde empiezan los posicionales

`getopt_long` recorre `argv`, consumiendo todas las opciones que encuentra (cortas y largas). Cuando termina, la variable global `optind` apunta al **primer elemento de `argv` que NO es opción**: el primer argumento posicional.

```
while ((opt = getopt_long(argc, argv, "hv:", ...)) != -1) {
    /* procesar opciones */
}

/* Aca optind == indice del primer argumento posicional */
printf("Posicionales:");
for (int i = optind; i < argc; i++) {
    printf(" %s", argv[i]);
}
```

Podés **consultar** `optind` en cualquier momento. Tampoco lo modificás vos; `getopt_long` lo va avanzando automáticamente a medida que consume opciones.

> ***Truco de laboratorio.*** *Podés verificar que* `optind` *funciona bien con este comando simple:*
>
> ```
> ./miapp -v --max=10 archivo.txt salida.txt
> ```
>
> *Cuando* `getopt_long` *termine,* `optind` *debería valer* `3`*, porque los posicionales empiezan en* `argv[3]` *(* `"archivo.txt"` *).*

5.4 optopt: la opcion desconocida

Cuando `getopt_long` encuentra una opción corta **que no está en el `optstring`**, guarda esa letra en `optopt`. Es útil para mostrar un mensaje de error más preciso:

```
case '?':
    fprintf(stderr, "Error: opcion desconocida '-%c'\n", optopt);
    break;
```

Si la opción desconocida es larga (`--desconocida`), `optopt` vale `0` . En ese caso, el mensaje de error por defecto de `getopt_long` (que ya imprime algo como `./prog: unrecognized option '--desconocida'`) ya es suficientemente claro.

5.5 getopt_reset(): reiniciar todo

Si necesitás pasar **otra vez** por los argumentos (por ejemplo, si llamaste a una función que también usa `getopt_long`), podés llamar a:

```
getopt_reset();
```

Esto resetea `optind` , `optarg` , y el estado interno para que la próxima llamada a `getopt_long` vuelva a recorrer desde el comienzo. En Windows con MSVC puro no está disponible (MinGW sí lo tiene). Si tu codigo corre en MSVC, usá esta alternativa:

```
optind = 0;
```

Es un hack sucio pero que funciona. `getopt_reset()` es solo eso: resetear `optind` a cero para que la próxima llamada recalcule donde empieza.

5.6 Por que globales? No es un bug, es un diseño

Las variables globas te pueden generar incomodidad: *Que pasa si dos hilos llaman a `getopt_long` al mismo tiempo?* --nada, porque `getopt_long` no es thread-safe y punto. Las herramientas de consola suelen ser monohebra. Usar globas para estas tres variables simplifica la API: no necesitás pasar un puntero a contexto por cada llamada; simplemente lo consultás cuando querés.

Es un intercambio clasico de diseno: **API mas simple ↔ no thread-safe**. Para una herramienta de consola, es el cambio correcto.

Práctica de la Parte 5

Ejercicio 13.7 -- Completar el uso de variables globales

Conceptos · `optarg` , `optind` , `optopt`

Contexto · Dada una llamada a `getopt_long` con un esqueleto incompleto, completar las líneas que usan las variables globales para mostrar los valores de las opciones. Simplificación: el programa es minimo; solo usa `optarg` .

Consigna · Completá las líneas marcadas con `/* ??? */` usando las variables globales correctas:

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

static struct option long_options[] = {
    {"help",    no_argument,       0, 'h'},
    {"output",  required_argument, 0, 'o'},
    {"verbose", no_argument,       0, 'v'},
    {0, 0, 0, 0}
};

int main(int argc, char *argv[])
{
    int opt;
    int option_index = 0;

    while ((opt = getopt_long(argc, argv, "ho:", long_options, &option_index)) != -1) {
        switch (opt) {
            case 'h':
                printf("Usando ayuda.\n");
                break;
            case 'o':
                printf("Archivo: ???\n");       /* que variable tiene el valor? */
                break;
            case 'v':
                printf("Verbose activo.\n");
                break;
        }
    }

    printf("Primer posicional: ???\n");     /* donde empiezan los posicionales? */

    return 0;
}
```

Entrada · El código incompleto arriba. **Salida** · Las dos líneas completadas con la variable global correcta y una breve justificación.

Criterios de aceptación

- [ ] Completaste `'o'` usando `optarg`.
- [ ] Completaste el posicional usando `argv[optind]`.
- [ ] Justificaste por qué cada variable es la correcta.

Ejercicio 13.8 -- Que vale optind despues de getopt_long?

Conceptos · `optind`, recorrido interno

Contexto · Entender **qué pasa adentro** cuando corré `getopt_long`. Simplificación: sin compilar, adivinar el valor de `optind` después de que `getopt_long` termina.

Consigna · Para cada invocación, escribí cuánto vale `optind` cuando `getopt_long` termina:

1. `./prog -v --max=10 archivo.txt salida.txt`
2. `./prog --help`
3. `./prog -hvo file.txt`

Entrada · Las tres invocaciones arriba. **Salida** · Tres valores de `optind`.

Criterios de aceptación

- [ ] Para la invocación 1, identificás que las opciones consumidas son `-v`, `--max=10`.
- [ ] Para la invocación 2, identificás que `--help` se consume y no quedan opcionales.
- [ ] Para la invocación 3, entendés que `-hvo` consume tres letras cortas.

PARTE 6 -- Errores y el caso `'?'`

6.1 Qué pasa cuando algo sale mal

`getopt_long` no deja que los errores pasen desapercibidos. Si el usuario hace algo incorrecto, la función **devuelve** `'?'` (el carácter punto de interrogación, valor ASCII 63) en lugar del código de una opción válida.

Hay tres situaciones que generan un retorno `'?'`:

Situación	Ejemplo	Qué muestra `getopt_long` por defecto
Opción corta desconocida	`-x` (si `-x` no está en el optstring)	`./prog: invalid option -- 'x'`
Opción larga desconocida	`--desconocida`	`./prog: unrecognized option '--desconocida'`
Valor faltante	`--max` sin valor (cuando requiere uno)	`./prog: option '--max' requires an argument`

En todos los casos, el retorno es `'?'`. Tu trabajo es manejarlo con estilo.

6.2 El caso `'?'` en el switch

El patrón estándar es simple:

```
case '?':
    /* Opción desconocida o valor faltante */
    fprintf(stderr, "Uso: %s [opciones] <archivo>\n", argv[0]);
    return 1;
```

Es un **error de entrada**: el usuario le pasó algo que tu programa no entiende. La respuesta correcta es **mostrar un mensaje de uso** y **terminar con código de error** (`return 1`).

> ***Ojo con esto.*** `getopt_long` ***ya imprimió*** *su propio mensaje de error por* `stderr` *antes de devolver* `'?'`*. Si volvés a imprimir el mismo error, vas a tener dos líneas. Podés:*
>
> 1. *Aceptar los dos mensajes (simplón).*
> 2. *Anular el mensaje de* `getopt_long` *implementando una función de error personalizada (avanzado, no lo necesitamos acá).*
> 3. *Ignorar el segundo mensaje y simplemente mostrar la ayuda (recomendado para este capítulo).*

6.3 El case default: por qué ponerlo siempre

Aunque ya cubriste todos los casos, poné siempre un `default:` en tu switch:

```c
switch (opt) {
    case 'h': /* ... */ break;
    case 'v': /* ... */ break;
    case 'o': /* ... */ break;
    case '?': /* error */ break;
    default:  /* nunca se llega aqui, pero esta aca por seguridad */ break;
}
```

Es una red de seguridad. Si en el futuro agregás un `case` nuevo y te olvidás de uno, al menos el programa **no se compila** (si tenés `-Wdefault` o similar). Con el `default`, la compilación pasa tranquila, sabiendo que cubriste todo.

Práctica de la Parte 6

Ejercicio 13.9 -- Manejar errores con `'?'`

Conceptos · caso `'?'`, `stderr`, códigos de salida

Contexto · Un programa que usa `getopt_long` necesita manejar el caso de opciones inválidas sin crashear. Simplificación: detectar la opcion desconocida y mostrar una ayuda.

Consigna · Completá el esqueleto siguiente para que maneje correctamente el caso `'?'` : cuando `getopt_long` devuelve `'?'` , el programa debe mostrar un mensaje de error por `stderr` con la línea de uso, e **inmediatamente retornar 1**.

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

static struct option long_options[] = {
    {"help",    no_argument,       0, 'h'},
    {"output",  required_argument, 0, 'o'},
    {0, 0, 0, 0}
};

int main(int argc, char *argv[])
{
    int opt;
    int option_index = 0;

    while ((opt = getopt_long(argc, argv, "ho:", long_options, &option_index)) != -1) {
        switch (opt) {
            case 'h':
                printf("Uso: %s [-h] [-o archivo]\n", argv[0]);
                return 0;
            case 'o':
                printf("Archivo: %s\n", optarg);
                break;
            case '?':
                /* COMPLETAR AQUI */
                ;
            default:
                break;
        }
    }
    return 0;
}
```

Entrada · El código incompleto arriba. **Salida** · El `case '?'` completado con el mensaje por `stderr` y el `return 1` .

Criterios de aceptación

- ☐ Escribís un `fprintf(stderr, ...)` con la línea de uso.
- ☐ Incluís `return 1;` inmediatamente después del `fprintf` .
- ☐ El código termina sin llegar al `default:` en caso de error.

PARTE 7 -- Posicionales despues de optind

7.1 Los posicionales son lo que queda

Llegó el momento más sencillo: una vez que `getopt_long` terminó de procesar todas las opciones (cortas y largas), los **argumentos posicionales** (archivos, nombres, datos "de verdad") quedan acumulados al final de `argv`, empezando en la posición `optind`.

```
/* despues de que el while termina, optind apunta al primer posicional */
for (int i = optind; i < argc; i++) {
    printf("Procesando: %s\n", argv[i]);
}
```

Es un bucle simple. No hay nada especial que hacer. `getopt_long` se encargó de **separar** las opciones (que consumió una por una) de los posicionales (que dejó intactos al final).

7.2 La convencion de -- (fin de opciones)

Existe una convención en la línea de comandos: un argumento que es **solo** `--` le dice a cualquier programa POSIX: *"de acá en adelante, todo es posicional, aunque empiece con guión"*.

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

Acá, `-archivo-que-empieza...` **no** se interpreta como una opción porque `--` ya marcó el límite.

`getopt_long` respeta esta convención automáticamente. Cuando ve un `--` solo, deja de buscar opciones y devuelve `-1`, dejando todo lo que sigue para `optind`.

7.3 Un ejemplo completo del flujo

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

static struct option long_options[] = {
    {"help",    no_argument,       0, 'h'},
    {"verbose", no_argument,       0, 'v'},
    {"output",  required_argument, 0, 'o'},
    {0, 0, 0, 0}
};

int main(int argc, char *argv[])
{
    int opt;
    int option_index = 0;
    int verbose = 0;
    const char *output_file = NULL;

    /* Primero: procesar todas las opciones */
    while ((opt = getopt_long(argc, argv, "ho:", long_options, &option_index)) != -1) {
        switch (opt) {
            case 'h':
                printf("Uso: %s [-hv] [-o archivo] <archivos>...\n", argv[0]);
                return 0;
            case 'v':
                verbose = 1;
                break;
            case 'o':
                output_file = optarg;
                break;
            case '?':
                fprintf(stderr, "Uso: %s [-hv] [-o archivo] <archivos>...\n", argv[0]);
                return 1;
        }
    }

    /* Segundo: procesar los posicionales (lo que quedaba) */
    for (int i = optind; i < argc; i++) {
        if (verbose) printf("Procesando: %s\n", argv[i]);
        /* ... aqui iria la logica real de procesamiento ... */
    }

    return 0;
}
```

Acá tenés el flujo completo en acción: primero opciones, luego posicionales. El mismo patrón se repite en **todos** los programas que usan `getopt_long`.

Práctica de la Parte 7

Ejercicio 13.10 -- Listar los posicionales despues de getopt_long

Conceptos · `optind`, bucle posicional, `argc`

Contexto · Dado un programa que procesó opciones con `getopt_long`, listar los archivos que quedaron. Simplificación: solo imprimirlos, sin procesar.

Consigna · Completá la parte final del siguiente programa para que liste todos los argumentos posicionales (desde `optind` hasta `argc-1`) después de que termina el bucle de opciones:

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

static struct option long_options[] = {
    {"help",    no_argument,       0, 'h'},
    {"verbose", no_argument,       0, 'v'},
    {"max",     required_argument, 0, 'm'},
    {0, 0, 0, 0}
};

int main(int argc, char *argv[])
{
    int opt;
    int option_index = 0;
    int verbose = 0;
    int max = 0;

    while ((opt = getopt_long(argc, argv, "hvm:", long_options, &option_index)) != -1) {
        switch (opt) {
            case 'h':
                printf("Uso: %s [-hv] [-m N] <archivos>...\n", argv[0]);
                return 0;
            case 'v':
                verbose = 1;
                break;
            case 'm':
                max = atoi(optarg);
                break;
            case '?':
                fprintf(stderr, "Uso: %s [-hv] [-m N] <archivos>...\n", argv[0]);
                return 1;
        }
    }

    /* COMPLETAR: listar los posicionales despues de este comentario */


    return 0;
}
```

El formato de salida para cada posicional debe ser: `Archivo N: <nombre>`, donde `N` es un contador que empieza en 1.

Entrada · El código incompleto arriba. **Salida** · El bucle completado que lista los posicionales.

Criterios de aceptación

- [ ] El bucle recorre desde `optind` hasta `argc - 1`.
- [ ] Cada posicional se imprime con el formato correcto.
- [ ] El contador empieza en 1, no en 0.

Ejercicio 13.11 -- Unir opciones y posicionales

Conceptos · flujo completo getopt_long + optind, variables locales

Contexto · Escribir un programa minimo pero completo que use `getopt_long` con una opcion corta y larga (con valor), procese los posicionales despues, e imprima un resumen. Simplificacion: no procesar ningun archivo de verdad, solo reportar lo que recibio.

Consigna · Escribí el programa completo desde cero. Debe aceptar:

Corta	Larga	Valor?	Accion
`-h`	`--help`	no	Muestra la ayuda y termina con 0.
`-n`	`--name`	si	Guarda el nombre del trabajo.

Y despues de procesar opciones, lista los posicionales (si hay). El formato de salida debe ser:

```
Trabajo: <nombre>
Opciones leídas: -n/-verbose, --help=0
Archivos a procesar: <cantidad>
  Archivo 1: <nombre>
  Archivo 2: <nombre>
```

Criterios de aceptación

- [ ] Incluye `<getopt.h>` y declara correctamente `long_options`.
- [ ] El optstring es `"hn:"`.
- [ ] La tabla tiene dos filas con terminador.
- [ ] `-h` / `--help` muestra ayuda y retorna 0.
- [ ] `-n` / `--name` guarda el valor de `optarg` en una variable local.
- [ ] El caso `'?'` muestra error por `stderr` y retorna 1.
- [ ] Despues del bucle, recorre desde `optind` hasta `argc` listando los posicionales.
- [ ] El formato de salida coincide con el ejemplo.

PARTE 7.5 -- La ayuda de uso: imprimir una guia clara

7.5.1 No hay funcion estandar en C para esto

Acá va la primera noticia: **C no tiene ninguna función estándar que genere automáticamente** una pantalla de ayuda formateada. Ni `printf`, ni `getopt_long`, ni nada en `<stdio.h>` o `<getopt.h>` lo hace por vos. Cada programa imprime su propia ayuda manualmente con `fprintf(stderr, ...)` o `printf(...)`.

Eso **no es un defecto**: es flexibilidad. Algunas herramientas muestran una ayuda corta de una línea; otras, una pantalla completa con descripciones detalladas. Vos decis cómo se ve.

Lo que sí existe es un **formato estándar de facto** que usan casi todas las herramientas POSIX — `ls`, `grep`, `tar`, `curl` — y que el usuario ya reconoce sin pensar. Aprender ese formato es tan importante como aprender a usar `getopt_long`.

7.5.2 El formato estandar: tu ejemplo como plantilla

Este es el formato canónico que vas a seguir. Es el mismo que usás en tu ejemplo, con ligeras mejoras:

```
Uso: mi_programa [opciones] <archivo_origen> <archivo_destino>
Argumentos posicionales:
  archivo_origen     Ruta del archivo que se va a leer (obligatorio, posición 1).
  archivo_destino    Ruta del archivo donde se guardará el resultado (obligatorio,
posición 2).

Opciones:
  -h, --help         Muestra esta ayuda y termina.
  -v, --verbose      Activa el modo detallado.
```

Desglosémoslo por secciones.

La línea de Uso

```
Uso: mi_programa [opciones] <archivo_origen> <archivo_destino>
```

Esta primera línea dice **todo** en un vistazo. Sigue reglas fijas de notación:

Notacion	Significado	Ejemplo
`[opcionales]`	Lo que va entre corchetes es **opcional** —se puede omitir.	`[-hv] [-o archivo]`
`<obligatorio>`	Los angulares indican un argumento **obligatorio**.	`<archivo_origen>`
`palabra`	Sin nada: lo que se escribe tal cual, como `--help`.	`--verbose`

La estructura típica es siempre:

```
Uso: <nombre_programa> [opciones] <posicionales_obligatorios>
```

NUNCA uses corchetes para envolver todo el uso —eso confunde. La línea debe ser tan clara que alguien la lea una vez y sepa cómo ejecutar el programa.

Argumentos posicionales

La sección de posicionales va **primero** (antes de las opciones) porque son la parte principal del programa: sin ellos, nada tiene sentido. Cada posicional lleva:

1. El nombre del argumento (en `snake_case`, minúsculas).
2. Un tabulador o dos espacios.
3. La descripción.
4. Entre paréntesis al final: si es obligatorio u opcional, y su posición numérica.

```
Argumentos posicionales:
  archivo_origen     Ruta del archivo que se va a leer (obligatorio, posición 1).
  archivo_destino    Ruta del archivo donde se guardará el resultado (obligatorio,
posición 2).
  modo               Modo de procesamiento: "normal" o "agresivo" (opcional, posición 3,
valor por defecto: "normal").
```

Opciones

La sección de opciones va **después**. Cada opción muestra la forma corta primero, luego la larga, separadas por coma. La descripción comienza en la misma columna que todas las demás descripciones:

```
Opciones:
  -h, --help         Muestra esta ayuda y termina.
  -v, --verbose      Activa el modo detallado.
  -o, --output <archivo>  Guarda el resultado en el archivo indicado.
  -m, --max <N>      Número máximo de elementos a procesar (por defecto: 100).
```

Notá que las opciones con valor incluyen `<nombre_del_valor>` entre la opción y la descripción. Las opciones sin valor van directo a la descripción.

7.5.3 Un helper para no repetir codigo

Aunque no hay función estándar, el patrón mas usado es crear una funcion propia llamada `print_help` que recibirá el nombre del programa (generalmente `argv[0]`) y escribirá toda la ayuda de golpe. Esto evita repetir el texto en cinco lugares distintos (caso `'?'` , caso `-h` , caso falta de argumentos, etc.) y garantiza que **siempre** se muestra el mismo texto.

```c
#include <stdio.h>

/* Declaramos la funcion antes de main: recibe el nombre del programa y muestra la ayuda
por stdout. */
static void print_help(const char *progname)
{
    printf("Uso: %s [opciones] <archivo_origen> <archivo_destino>\n", progname);
    printf("Argumentos posicionales:\n");
    printf("  archivo_origen     Ruta del archivo que se va a leer (obligatorio, posicion
1).\n");
    printf("  archivo_destino    Ruta del archivo donde se guardara el resultado
(obligatorio, posicion 2).\n");
    printf("\n");
    printf("Opciones:\n");
    printf("  -h, --help         Muestra esta ayuda y termina.\n");
    printf("  -v, --verbose      Activa el modo detallado.\n");
    printf("  -o, --output <archivo>  Guarda el resultado en el archivo indicado.\n");
    printf("  -m, --max <N>      Numero maximo de elementos a procesar (por defecto:
100).\n");
}
```

Despues, la llamás desde cualquier lugar donde necesites mostrar ayuda:

```
case 'h':
    print_help(argv[0]);
    return 0;

case '?':
    fprintf(stderr, "Error: opcion invalida.\n\n");
    print_help(argv[0]);
    return 1;
```

> ***Momento ajá.*** *La ventaja de* `print_help` *no es solo que escribís menos código: es que si mañana agregás una opción nueva, la modificás en un solo lugar (* `print_help` *) y* ***todas*** *las salidas se actualizan automáticamente. Sin ella, tendrías que buscar y reemplazar cada aparición del texto de ayuda —y muy probablemente te olvidás de una.*

7.5.4 stdout vs stderr para la ayuda

La ayuda tiene un comportamiento mixto:

- Cuando el usuario **pide** la ayuda explícitamente (`-h` o `--help`): va por `stdout` . El usuario pidió ver algo; que salga en pantalla sin interferir con redirecciones es lo correcto.
- Cuando la ayuda aparece como **consecuencia de un error** (opción desconocida, falta de argumentos): va por `stderr` , porque está acompañando un mensaje de error.

Esto permite dos flujos distintos:

```
./mi_programa --help > ayuda.txt    # el usuario guardó la ayuda en un archivo
./mi_programa --desconocida         # el error con ayuda aparece en pantalla, no se pierde
```

Práctica de la Parte 7.5

Ejercicio 13.12 -- Crear la funcion print_help

Conceptos · `printf`, estructura de ayuda, separación stdout/stderr

Contexto · Dado un conjunto de opciones y posicionales, escribir la función `print_help` que los muestre con el formato estándar del capítulo. Simplificacion: no escribís el programa principal, solo la funcion de ayuda.

Consigna · Escribí una función `print_help(const char *progname)` para un programa que tiene:

Corta	Larga	Valor?	Descripcion
`-h`	`--help`	no	Muestra esta ayuda y termina.
`-q`	`--quiet`	no	Suprime los mensajes informativos.
`-d`	`--database`	si	Especifica la base de datos a conectar.

Y que espera dos argumentos posicionales:

1. `archivo_conf` — Ruta del archivo de configuración (obligatorio, posición 1).
2. `archivo_log` — Ruta del archivo de registro de eventos (opcional, posición 2, valor por defecto: `/dev/null`).

El formato debe seguir **exactamente** la plantilla de la sección 7.5.2, con las secciones en este orden: Uso → Argumentos posicionales → Opciones.

Entrada · Los datos arriba. **Salida** · La función C completa `print_help`.

Identificadores sugeridos · `progname`

Criterios de aceptación

- ☐ La línea de Uso usa `progname` y la notación correcta (`<>` , `[]`).
- ☐ Los posicionales se listan con sus descripciones completas, incluyendo obligatorios/opcionales y posiciones.
- ☐ Las opciones muestran ambas variantes (corta/larga) separadas por coma.
- ☐ Las opciones con valor incluyen `<nombre>` entre la opción y la descripción.
- ☐ Hay una línea en blanco entre la sección de posicionales y la de opciones.

Ejercicio 13.13 -- Usar print_help en el programa completo

Conceptos · `getopt_long`, `print_help`, `stdout` / `stderr`, flujo de ayuda

Contexto · Integrar `print_help` en un programa real con `getopt_long`. Simplificacion: el programa solo muestra la ayuda y termina; no procesa ningun dato.

Consigna · Escribí un programa completo que:

1. Use `getopt_long` con las opciones `-h`, `-q`, `-d <database>`.
2. Llame a `print_help(argv[0])` (definida por vos siguiendo el formato estándar) cuando se detecta `-h` / `--help`, retornando 0.
3. Muestre `print_help(argv[0])` por `stderr` y retorne 1 cuando `getopt_long` devuelve `'?'`.
4. Si no recibe el argumento posicional obligatorio (`archivo_conf`), muestre la ayuda por `stderr` con un mensaje de error antes y retorne 1.
5. Si todo está bien, imprima: `Conectando a base de datos: <valor_de_d> desde <archivo_conf>` por `stdout` y retorne 0.

Escenario de prueba

```
$ ./conector --help
Uso: ./conector [opciones] <archivo_conf> [archivo_log]
Argumentos posicionales:
  archivo_conf     Ruta del archivo de configuración (obligatorio, posición 1).
  archivo_log      Ruta del archivo de registro (opcional, posición 2, por defecto:
/dev/null).

Opciones:
  -h, --help               Muestra esta ayuda y termina.
  -q, --quiet              Suprime los mensajes informativos.
  -d, --database <database> Especifica la base de datos a conectar.
```

```
$ ./conector
Error: se requiere un archivo de configuración
Uso: ./conector [opciones] <archivo_conf> [archivo_log]
... (resto de la ayuda por stderr)
```

```
$ ./conector -d mi_db config.ini
```

```
Conectando a base de datos: mi_db desde config.ini
```

Criterios de aceptación

- [ ] `print_help` existe como funcion estatica y se llama desde `-h`, `'?'`, y falta de argumentos.
- [ ] La línea de Uso coincide exactamente con el formato canónico.
- [ ] El caso `'?'` imprime un error por `stderr` ANTES de mostrar la ayuda (tambien por `stderr`).
- [ ] `-h` muestra todo por `stdout` y retorna 0.
- [ ] Falta de posicional obligatorio muestra error + ayuda por `stderr`, retorna 1.
- [ ] Flujo exitoso imprime el mensaje correcto por `stdout`, retorna 0.

PARTE 8 -- El ejercicio jefe

8.1 La prueba final

Llegó el momento de juntar todo lo aprendido en un solo programa. Este no es un ejercicio cualquiera: es una **herramienta de consola completa** que usa `getopt_long` para interpretar opciones cortas, opciones largas con valores, y deja los posicionales intactos al final.

Ejercicio 13.14 -- Utilidad de consola completa (JEFE)

Conceptos · `getopt_long`, `struct option`, `optarg`, `optind`, `stderr`, códigos de salida, variables locales, bucle posicional, `print_help`.

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 usando `getopt_long`:

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. Defina una funcion `static void print_help(const char *progname)` siguiendo el formato estándar de la Parte 7.5.
2. Use `print_help(argv[0])` tanto para `-h` / `--help` (por `stdout`, retorno 0) como para errores por `'?'` (por `stderr`, retorno 1).
3. Acepte las **dos formas** de valor para las largas (`--output=x` y `--output x`). Esto lo resuelve `getopt_long` automáticamente --no necesitás hacer nada extra.
4. Trate los argumentos **no-opción** como archivos posicionales (listados al final usando `optind`).
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
Maximo: 10
Archivos a procesar: 2
  Archivo 1: entrada.txt
  Archivo 2: salida.txt
```

```
$ ./util -v --max=5 datos.csv
Modo verbose: ON
Maximo: 5
```

```
Archivos a procesar: 0
Aviso: no hay archivos de entrada ni archivo de salida definido
```

```
$ ./util --help
Uso: ./util [opciones] <archivos>...
Argumentos posicionales:
  archivos       Lista de archivos a procesar (al menos uno obligatorio).

Opciones:
  -h, --help               Muestra esta ayuda y termina.
  -v, --verbose            Activa el modo detallado.
  -o, --output <archivo>   Archivo de salida.
  -m, --max <N>            Maximo de elementos (por defecto: 100).
```

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

Restricciones / pistas

- Usá `getopt_long` con el `optstring` `"hvo:m:"` .
- **No** olvides el terminador `{0,0,0,0}` de la tabla.
- El `-h` tiene **prioridad**: mostralo apenas aparece y terminá.
- Los posicionales salen de `argv[optind..argc-1]` .
- Errores → `stderr` ; resultados → `stdout` .
- Si no hay archivos ni output definido, mostrá un aviso (no es error fatal, pero sí informativo).

Criterios de aceptación

- ☐ Reconoce `-v` / `--verbose` , `-o` / `--output` y `-m` / `--max` .
- ☐ Acepta valor pegado (`--max=10`) y separado (`--max 10`).
- ☐ `print_help` existe como funcion estatica con el formato canónico.
- ☐ `-h` / `--help` muestra ayuda por `stdout` y retorna 0.
- ☐ Una opción desconocida produce error por `stderr` (con help incluido) y código != 0.
- ☐ Los posicionales se listan después de las opciones (usando `optind`).
- ☐ El formato coincide con los escenarios de prueba.
- ☐ Sin archivos ni output, muestra un aviso informativo.

Cierre: lo que te llevás

Si tuvieras que quedarte con **una sola idea**, que sea esta: `getopt_long` hace todo el trabajo pesado de parsear opciones para vos --vos solo declarás qué opciones reconocés y qué querés hacer con cada una.

- **La llamada es siempre la misma.** `getopt_long(argc, argv, "hvo:m:", long_options, &option_index)` . Cambiá el optstring, cambiá la tabla, pero el patrón no cambia.
- `struct option` **tiene cuatro campos.** Nombre sin guiones, macro de argumento, `0` , código corto. Y el terminador `{0,0,0,0}` al final --siempre.
- **Tres macros definen si lleva valor.** `no_argument` (flag), `required_argument` (lleva siempre un valor), `optional_argument` (puede llevar o no).
- **Tres variables globales hacen el trabajo sucio.** `optarg` tiene el valor de la opción, `optind` apunta al primer posicional, `optopt` tiene la letra de la opción desconocida.
- `optind` **es tu puente hacia los datos reales.** Después del bucle `while` , un `for` desde `optind` hasta `argc` te da todos los posicionales --archivos, nombres, lo que importa.
- **El caso** `'?'` **captura todos los errores.** Si la opción es desconocida o falta un valor, `getopt_long` devuelve `'?'` . Manejalos mostrando ayuda por `stderr` y retornando 1.
- **La ayuda se imprime con un helper propio.** No existe funcion estandar en C para generarla automaticamente; el patrón es crear una `print_help(const char *progname)` que centralice el texto y lo muestre siguiendo el formato canónico (Uso → posicionales → opciones).

Y sobre todo: aprendiste a **delegar**. Escribir un parser de opciones desde cero es posible (y lo hiciste en el capítulo anterior), pero `getopt_long` lo hace mejor, más consistente, y con menos líneas de código. Eso no es trampa --es programar inteligente.