Capítulo 21 — Makefiles: de "compilar a mano" a proyectos profesionales

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en funciones (`05`), arreglos y cadenas (`06`), punteros (`07`), structs (`08`), memoria dinámica (`09`) y archivos (`10`). También asume que ya compilaste un programa con `gcc` .*
>
> *Este capítulo no es solo sobre `make` : es sobre **automatizar tu flujo de trabajo**. Cada vez que repetís el mismo comando dos veces, algo anda mal. Acá vas a aprender a dejar que una herramienta se encargue de compilar, enlazar y construir bibliotecas.*

Cómo leer este capítulo

Este es el capítulo donde "compilar a mano" se convierte en "**la máquina compila por mí**". Ya no vas a escribir la línea de gcc cada vez. Vas a escribir **las reglas** y dejar que una herramienta lo haga.

Está dividido en **ocho partes**. Después de cada concepto importante hay uno o varios ejercicios cortos. **No los saltees**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	El problema: compilar a mano	Cada cambio obliga a recompilar todo con el mismo comando largo: rápido, repetitivo, propenso a error.
2	Makefile básico	`make` ejecuta tus reglas: variables, targets, dependencias y recipes.
3	Patrones y automático	`%`, `$@`, `$^`: escribir una regla que sirve para cualquier `.c`.
4	Variables avanzadas	`+=`, `-=`, `?=` : configurar tu build sin copiar-pastear.
5	Bibliotecas estáticas	`ar` y `ranlib`: empaquetar objetos en un archivo que el linker usa como si fuera código.
6	Bibliotecas dinámicas	`gcc -shared -fPIC`: crear `.so` / `.dll` que se cargan al ejecutar, no al compilar.
7	Makefiles profesionales	Include condicional, sanitizers, targets implícitas y variables integradas de gcc.
8	Integrador multi-subdirectorio	Proyectos con `lib/`, `tests/` y `$(MAKE) -C <dir>` para orquestar builds anidados.
9	Instalación y configuraciones	`PREFIX`, `make install/uninstall`, feature toggles (`RENDERER=sdl

Los **ejercicios** se numeran en orden: `21.1`, `21.2`, `21.3`, y así. Cada uno trae su **escenario de prueba** (qué hacés y qué tiene que pasar) para que lo ejecutes y compruebes.

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

PARTE 1 — El problema: compilar a mano

Empecemos por lo que ya hacés (o harías) hoy. Y por qué eso se vuelve un problema rápido.

1.1 "Compilar con gcc" no es una vez, es siempre

La primera vez que compilas algo, el comando parece inocente:

```
$ gcc main.c -o programa
```

Funciona. Pero ahora imaginá esto: tenés un proyecto con **diez archivos** `.c` y tres de ellos cambiaron desde la última compilación. ¿Qué hacés?

Opción A — recompilar todo:

```
$ gcc main.c file1.c file2.c utils.c helper.c parser.c config.c network.c io.c mathops.c
\
    -o programa -Wall -Wextra -I./include -L./lib -lmylib -lm
```

Opción B — recompilar solo lo que cambió (si te acordás cuáles):

```
$ gcc main.c file1.c file2.c utils.c helper.c parser.c config.c network.c io.c mathops.c
\
    -o programa -Wall -Wextra -I./include -L./lib -lmylib -lm
```

Sí, misma opción. Porque **gcc no sabe qué archivos cambiaron**: vos tenés que decirle cuáles recompilar y cuáles reusar. Y la línea de arriba tiene **doce archivos**, **seis banderas**, **tres flags de linker**. Si te olvidás de uno, el programa no compila bien. Si escribís mal `-Wall`, no ves advertencias nuevas.

Y esto es lo peor: **cada vez que hagas un cambio**, tenés que ejecutar esa línea otra vez. O una variante si querés optimizar para producción (`-O2`). O con sanitizers para debugging (`-fsanitize=address,undefined`). El comando se multiplica y el error humano entra por la puerta grande.

> ***La analogía del chef.*** *Compilar un programa a mano es como ser chef, lavaplatos y mozo al mismo tiempo. Cada vez que cocinás algo nuevo, tenés que recordar exactamente qué ingredientes usaste, en qué orden, con qué fuego, y repetir el proceso cada vez que alguien pida lo mismo. Un sistema de build es como un* ***chef asistente*** *que se acuerda de todo: "solo los platos que cambiaron necesitan prepararse de nuevo".*

1.2 ¿Por qué `make`? ¿No hay alternativas?

Sí las hay: `cmake`, `ninja`, `bazel`, `meson`, `just`. Pero `make` es **universal**: está en Linux, macOS (con un `brew install make`), Windows (MSYS2/MinGW). Es el **estándar mínimo** que todo programador C debería conocer. Las otras herramientas son capas encima de `make` o para proyectos más grandes:

```
cmake  →  genera Makefiles (o Ninja, o Xcode, o Visual Studio projects)
ninja  →  ejecuta reglas muy rápido (usa un formato propio)
bazel  →  multiproyecto, multi-lenguaje, con cache de build
just   →  reemplazo moderno de make con syntax más limpia
```

Conocer `make` no significa que siempre uses `make`. Significa que **entendés el concepto de "reglas de build"**, que es la idea central. Las otras herramientas comparten esa misma idea: tenés **dependencias**, un **target** y una **receta**. Si entendés eso, podés migrar a cualquier otra cosa.

1.3 La filosofía de `make`

Antes de escribir la primera regla, entiende la **pregunta central** que `make` se hace:

> ***"¿Este target necesita reconstruirse?"***

Y responde con un solo algoritmo:

```
    ¿el archivo del target existe?          →  no   →  ejecutar la receta
    ¿la fecha de alguna dependencia es más reciente que el target? →  sí → ejecutar la
receta
    ¿todas las dependencias son más viejas? →  no   →  SKIP (ya actualizado)
```

Es **todo o nada**: si una dependencia cambió, reconstruís. Si no, no hacés nada. Este "no hacer nada" es lo que hace a `make` tan rápido en proyectos grandes: solo se toca lo necesario.

> ***Eso de "fecha más reciente"*** *es la clave.* `make` *no sabe qué código cambiaste; mira los* ***timestamps*** *de los archivos. Si modificaste* `main.c` *, su timestamp se actualizó. Si el ejecutable tiene un timestamp anterior,* `make` *decide que hay que recompilar. Simple y efectivo.*

PARTE 2 — Makefile básico: tu primera regla

Empezamos con lo más simple posible. Un solo archivo `.c` , un comando de compilación. Sin optimizaciones, sin sanitizers. El equivalente a `gcc main.c -Wall -o programa` pero dentro de un Makefile.

2.1 La estructura mínima de un Makefile

Un **Makefile** (o `makefile` , que es lo mismo) es un archivo de texto con una sintaxis muy estricta:

```
target : dependencias
    <receta>            ← ¡ESTE SPACIO ES UN TABULADOR, NO ESPACIOS!
```

Cada línea de receta **debe empezar con un carácter TAB**. No espacios. Si usás espacios, `make` va a decirte:

```
Makefile:3: *** missing separator.  Stop.
```

Ese error es el primero que todo el mundo ve y **no significa** "tenés un error de sintaxis": significa "la línea de arriba empieza con espacios, poné un TAB".

Acá tenés tu primer Makefile, llamado `Makefile` (sin extensión):

```
CC = gcc
CFLAGS = -Wall -Wextra
TARGET = programa
SRCS = main.c utils.c parser.c
OBJS = $(SRCS:.c=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
[TAB]$(CC) $(OBJS) -o $(TARGET)

%.o: %.c
[TAB]$(CC) $(CFLAGS) -c $< -o $@

clean:
[TAB]rm -f $(OBJS) $(TARGET)
```

Vamos a desglosar cada línea. **No es mucho**: son cuatro conceptos y seis líneas de receta.

Concepto 1: variables

```
CC = gcc
CFLAGS = -Wall -Wextra
```

`CC` es la variable del compilador (por convención; `make` ya trae `CC = gcc` como default, pero es bueno declararlo explícitamente). `CFLAGS` es donde van las banderas del compilador. La razón por la que las declarás como variables: **podés cambiarlas todas desde un solo lugar**, y también **sobreescribirlas desde la línea de comandos**:

```
$ make CFLAGS="-Wall -Wextra -O2"    # sobreescribís CFLAGS para este invocation
$ CC=clang make                      # usás clang en vez de gcc
```

Concepto 2: target `all`

```
all: $(TARGET)
```

`all` es el **target por defecto**. Cuando ejecutás `make` sin argumentos, ese es el primer target que se construye. Es una convención universal: si alguien ve tu Makefile, espera encontrar un `all`.

Concepto 3: la regla de compilación

```
$(TARGET): $(OBJS)
[TAB]$(CC) $(OBJS) -o $(TARGET)
```

Esta regla dice: "**para construir** `$TARGET` (que es el ejecutable), necesito los objetos en `$OBJS`. Si cualquier `.o` está más nuevo que el `.o`, ejecutá la receta." La receta es el linker: junta todos los `.o` y produce el ejecutable.

Concepto 4: la regla de patrón

```
%.o: %.c
[TAB]$(CC) $(CFLAGS) -c $< -o $@
```

Esta es la **regla de patrón**: dice "cualquier `.o` se construye a partir de un `.c` del mismo nombre, con estas banderas". `%` es un comodín: si el target es `main.o`, `%` es `main`; si es `utils.o`, `%` es `utils`.

Y `$<` y `$@` son **variables automáticas** (las vemos en detalle más abajo):

Variable	Qué significa
`$@`	El target de la regla (`$@` = `main.o`)
`$<`	La primera dependencia (`$<` = `main.c`)
`$^`	Todas las dependencias separadas por espacio

Con estas tres variables automáticas, **una sola regla de patrón te sirve para cualquier archivo `.c`** del proyecto. No necesitás escribir una regla por cada uno.

Concepto 5: `clean`

```
clean:
[TAB]rm -f $(OBJS) $(TARGET)
```

`clean` **no es un target real**: no produce ningún archivo que se llame `clean`. Es un target "de mantenimiento", una orden manual. Por eso existe `.PHONY` (lo vemos en la Parte 7): le dice a `make` que `clean` no genera un archivo, así si alguien crea un archivo llamado `clean` por error, `make` igual ejecuta la receta.

La regla de los archivos: `SRCS` y `OBJS`

```
SRCS = main.c utils.c parser.c
OBJS = $(SRCS:.c=.o)
```

La segunda línea usa **sustitución de patrones**: reemplaza `.c` por `.o` en la lista de `SRCS`. El resultado es:

```
    OBJS = main.o utils.o parser.o
```

Esto es **automático**: si agregás un nuevo archivo a `SRCS`, `OBJS` se actualiza solo. No tenés que tocar nada más.

2.2 Cómo se usa `make`

```
$ make                # construye el primer target (all → programa)
$ make clean          # borra objetos y ejecutable
$ make CFLAGS="-Wall -O2"   # sobreescribís las banderas solo para este invocation
```

Cuando ejecutás `make`, ve qué archivos cambiaron, recompila lo que hace falta y enlaza todo. Si no cambió nada:

```
$ make
make: 'programa' is up to date.
```

> ***Momento ajá.*** *Esa frase —"is up to date"— es el momento más satisfactorio de la programación. Es la máquina diciéndote "no hice nada porque no había que hacer nada". En proyectos grandes, eso puede significar "30 segundos de ahorro en vez de 5 minutos de compilación".*

Práctica de la Parte 2

Ejercicio 21.1 — El Makefile mínimo

Conceptos · variables (`CC`, `CFLAGS`), target `all`, regla de patrón, `$@`, `$<`, `clean`

Contexto · Tenés un programa simple con tres archivos: `main.c`, `utils.c` y `parser.c`. Cada uno tiene su propia función. El ejecutable se llama `palabra`. Es una simplificación: no hay includes ni librerías externas.

Consigna · Creá un archivo `Makefile` en el mismo directorio que tus fuentes, con:

1. Variables `CC`, `CFLAGS = -Wall -Wextra`, `TARGET = palabra`, y `SRCS` listando los tres archivos `.c`.
2. Target `all` que depende de `$(TARGET)`.
3. La regla de enlace (`$(TARGET): $(OBJS)`).
4. La regla de patrón para `.o → .c`.
5. El target `clean`.

Después:

```
$ make clean   # borra cualquier cosa vieja
$ make         # compila todo desde cero
$ ./palabra    # ejecutá el resultado
```

Entrada · Ninguna (el programa imprime algo al arrancar). **Salida** · El Makefile funciona: `make` compila y enlaza sin errores, y `make clean` borra todo.

Escenario de prueba

```
$ make
gcc -Wall -Wextra -c main.c -o main.o
gcc -Wall -Wextra -c utils.c -o utils.o
gcc -Wall -Wextra -c parser.c -o parser.o
gcc main.o utils.o parser.o -o palabra

$ ./palabra
Hola desde palabra

$ make         # segunda vez: nada que hacer
make: 'palabra' is up to date.

$ make clean
rm -f main.o utils.o parser.o palabra
```

Identificadores sugeridos · `Makefile` , `TARGET` , `SRCS` , `OBJS`
Restricciones/pistas

- Usá `$@` y `$<` en las reglas de patrón. No escribas los nombres a mano.
- `clean` debe borrar todos los `.o` **y** el ejecutable.

Criterios de aceptación

- ☐ `make` compila los tres archivos y genera el ejecutable sin errores.
- ☐ `make` segunda vez dice "is up to date" (no recompila nada).
- ☐ `make clean` borra todos los `.o` y el ejecutable.
- ☐ Se usa `$@` y `$<` en la regla de patrón.

Ejercicio 21.2 — El error del espacio vs tabulador

Conceptos · sintaxis de Makefile, caracteres de tabulación, diagnóstico del error "missing separator"

Contexto · Este es el primer y más común error al escribir un Makefile. Si lo entendés desde ya, no te va a frustrar la primera vez (que sí va a pasar).

Consigna · Creá un Makefile con **espacios en vez de TABs** en las recetas (copiá el ejercicio 21.1 pero reemplazá los TAB por cuatro espacios). Compilalo con `make` y **leé el mensaje de error completo**. Después corregilo cambiando los espacios por un TAB real y comprobá que funcione.

Escribí un párrafo corto explicando:

1. Cuál fue el mensaje exacto.
2. Por qué `make` exige un TAB y no permite espacios (una razón histórica o de parsing).
3. Cómo podés asegurarte de que tu editor usa TABs en los Makefiles.

Entrada · Ninguna. **Salida** · El Makefile corregido + párrafo explicativo.

Escenario de prueba

```
$ make
Makefile:8: *** missing separator.  Stop.
```

Restricciones/pistas

- La mayoría de los editores modernos permiten configurar "insert spaces instead of tabs" por tipo de archivo. Usá esa configuración para `Makefile`.
- Si tu editor no lo soporta, podés verificar el Makefile con:

  ```
  $ cat -A Makefile | grep '^\s'   # los TAB aparecen como ^I
  ```

Criterios de aceptación

- [ ] El mensaje de error está citado textualmente.
- [ ] La explicación del "por qué TAB" es correcta (historia o parsing).
- [ ] Se documentó al menos una forma de prevenir el error en tu editor.
- [ ] El Makefile corregido funciona: `make` compila sin errores.

PARTE 3 — Patrones y variables automáticas

Ya escribiste un Makefile que funciona para tres archivos. Pero ¿qué pasa cuando tenés **cincuenta**? Acá entran las reglas de patrón y las variables automáticas, que son la razón por la que `make` no se vuelve un archivo de 2000 líneas.

3.1 La regla de patrón: una para todas

La regla de patrón usa `%` como comodín:

```
%.o: %.c
[TAB]$(CC) $(CFLAGS) -c $< -o $@
```

Esto le dice a `make` : "**siempre** que necesite construir un archivo que termine en `.o` a partir de un archivo con el mismo nombre pero `.c` , usá esta receta." No importa cuántos archivos `.c` tengas: una regla de patrón los cubre todos.

Y hay reglas de patrón para **cualquier extensión**:

```
%.o: %.s          # ensamblador → objeto
[TAB]$(CC) $(ASFLAGS) -c $< -o $@

%.html: %.md       # markdown → html (para documentación, por ejemplo)
[TAB]pandoc $< -o $@
```

3.2 Las variables automáticas completas

`$@` , `$<` y `$^` son las tres más usadas. Pero hay más:

Variable	Qué significa	¿Cuándo usarla
`$@`	El target de la regla	Siempre que necesites saber "¿qué estoy construyendo?"
`$<`	La primera dependencia (ordenada)	Cuando necesitás solo una dependencia (ej. un `.c`)
`$^`	**Todas** las dependencias, ordenadas y sin duplicados	Cuando necesitás listar todo (ej. varios `.o` para enlazar)
`$?`	Dependencias **más recientes** que el target	Para compilar solo lo que cambió dentro de la misma receta
`$*`	La parte del comodín (`%`)	Cuando necesitás saber "¿qué archivo es?" (ej. `main.o` → `main`)

Ejemplo con `$^` en la regla de enlace:

```
$(TARGET): $(OBJS)
[TAB]$(CC) $^ -o $@ $(LDLIBS)
```

Ejemplo con múltiples dependencias

Imaginate una regla donde el target tiene **varias** dependencias:

```
programa: main.o utils.o parser.o libmath.a
[TAB]$(CC) $^ -o $@ -lm   # $^ = main.o utils.o parser.o libmath.a
```

`$^` expande a `main.o utils.o parser.o libmath.a` — **todas las dependencias**. `$?` sería solo las más recientes. Y `$@` es `programa`.

3.3 Reglas implícitas: `make` ya sabe mucho

Antes de escribir reglas, sabé que `make` tiene **reglas implícitas** (también llamadas "built-in rules") para compilaciones comunes. Por ejemplo, `make` **ya sabe** que para construir `main.o` necesita `main.c`:

```
    Regla implícita de make:
    %.o: %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
```

Esto significa que, en su forma más simple, podés tener un Makefile **vacío**:

```
# ¡Makefile vacío! make igual compila todos los .o y enlaza.
all: programa
programa: main.o utils.o parser.o
[TAB]$(CC) $^ -o $@
```

O incluso, si solo tenés un `main.c`:

```
$ touch Makefile   # Makefile vacío
$ make             # make usa la regla implícita para main.o y después enlaza
```

Pero las reglas implícitas **no usan** tus variables personalizadas (`CFLAGS`, etc.) ni tus banderas custom. Por eso siempre es mejor escribir las reglas explícitamente: tenés control total.

> ***Para guardar.*** *Las reglas implícitas son un "escape hatch" para proyectos triviales de un solo archivo, pero en cualquier proyecto real* ***escribí las reglas explícitas****. Porque necesitás* `-Wall`, `-Wextra`, `-I./include` *y* `$(LDFLAGS)`.

3.4 El doble punto: dependencias sin archivo

Ya vimos `.PHONY`. Pero hay otros "targets especiales" que empiezan con `.` y no generan un archivo:

Target	Qué significa
`.PHONY`	Lista de targets que **no son archivos** (como `clean`, `all`, `install`)
`.SUFFIXES`	Lista de extensiones reconocidas (para reglas implícitas)
`.DEFAULT`	Target por defecto si no se encuentra ninguno (raro, casi nunca usado)

`.PHONY` es importante: le dice a `make` que esos targets **siempre deben ejecutarse**, incluso si existe un archivo con ese nombre. Si no lo declarás y alguien crea un archivo llamado `clean`, `make` no va a ejecutar la receta (porque cree que ya está "actualizado").

```
.PHONY: all clean install
```

> ***La trampa del Makefile sin*** `.PHONY`***.*** *Es el bug silencioso más común: creás un archivo llamado* `clean`, `all` *o* `install` *(por accidente, por un script, por copiar-paste), y de repente* `make clean` *no hace nada. La solución:* ***declarar siempre*** `.PHONY`.

Práctica de la Parte 3

Ejercicio 21.3 — Reglas de patrón para ensamblador

Conceptos · reglas de patrón, `$@`, `$<`, `$^`, variables automáticas

Contexto · A veces necesitás compilar código en ensamblador (por performance o porque una librería externa lo distribuye así). Hacés las reglas necesarias para que `make` compile `.s → .o`.

Consigna · Tenés un proyecto con:

- `main.c` — código C principal
- `utils.c` — código C auxiliar
- `vector.s` — una función en ensamblador (asumí que tiene una función exportada `vec_add`)

Creá las reglas necesarias para compilar todo y generar el ejecutable `mixto`:

1. Regla de patrón `%.o: %.c`
2. Regla de patrón `%.o: %.s` con una variable `ASFLAGS`
3. Regla de enlace que usa `$^`

Asegurate de declarar `.PHONY: all clean`.

Entrada · Los archivos fuente (`main.c`, `utils.c`, `vector.s`). **Salida** · El Makefile completo y el ejecutable `mixto` funcionando.

Escenario de prueba

```
$ make
gcc -Wall -Wextra -c main.c -o main.o
gcc -Wall -Wextra -c utils.c -o utils.o
nasm -f elf64 -g -F dwarf vector.s -o vector.o
gcc main.o utils.o vector.o -o mixto

$ ./mixto
Vector sum result: 42
```

Identificadores sugeridos · `Makefile`, `ASFLAGS`, `MIXTO` **Restricciones/pistas**

- No necesitás escribir el ensamblador real: asumí que existe y tiene la función `vec_add`.
- Usá `$^` en la regla de enlace.

Criterios de aceptación

- [ ] Existe una regla de patrón para `.s → .o`.
- [ ] Se usa `$^` en la regla de enlace.
- [ ] `.PHONY` incluye `all` y `clean`.
- [ ] El Makefile compila todo y genera el ejecutable.

Ejercicio 21.4 — Entender las variables automáticas

Conceptos · `$@`, `$<`, `$^`, `$?`, `$*`, diagnóstico con `echo`

Contexto · Las variables automáticas son poderosas, pero si no sabés qué devuelve cada una, escribís recetas que parecen magia. Este ejercicio te obliga a **ver lo que hacen**.

Consigna · Escribí un Makefile con **una regla de patrón simple** y otra regla con múltiples dependencias. En cada receta, usá `echo` para imprimir el valor de **cada variable automática**:

```
%.o: %.c
[TAB]@echo "[$@] target"
[TAB]@echo "[$<] first prerequisite"
[TAB]@echo "[$^] all prerequisites"
[TAB]@echo "[$?] newer prerequisites"
[TAB]@echo "[$*] stem (pattern match)"
[TAB]$(CC) $(CFLAGS) -c $< -o $@

programa: main.o utils.o parser.o
[TAB]@echo "[$@] target: programa"
[TAB]@echo "[$^] all prereqs: $(words $^) items"
[TAB]$(CC) $^ -o $@
```

El `@` al inicio de la línea evita que `make` imprima el comando antes de ejecutarlo (solo ves el resultado del `echo`).

Ejecutá `make clean && make` y **documentá la salida completa**: qué imprimió cada variable, y por qué.

Entrada · Un Makefile con `echo` en las recetas + algunos `.c` . **Salida** · La salida de `make` , línea por línea, explicada.

Escenario de prueba

```
$ make clean && make
rm -f *.o programa
gcc -Wall -Wextra -c main.c -o main.o
[main.o] target
[main.c] first prerequisite
[main.c] all prerequisites
[main.c] newer prerequisites
[main] stem (pattern match)

...
```

Restricciones/pistas

- El `@` al inicio de la línea hace que `make` **no imprima el comando**, solo el resultado del `echo` . Sin el `@` , cada `echo` se imprime dos veces: una vez el comando mismo y otra su salida.

Criterios de aceptación

- ☐ Cada variable automática (`$@` , `$<` , `$^` , `$?` , `$*`) se imprime y explica.
- ☐ La regla con múltiples dependencias muestra cómo `$^` los junta todos.
- ☐ Se nota la diferencia entre `$?` (solo las nuevas) y `$^` (todas).

PARTE 4 — Variables avanzadas de Makefile

Ya sabés que `CC = gcc` y `CFLAGS = -Wall` . Pero un Makefile real tiene **decenas** de variables, y saber cuáles usar y cómo combinarlas es lo que separa un Makefile básico de uno profesional.

4.1 Tipos de asignación

Hay cuatro formas de asignar variables, y cada una tiene un comportamiento distinto:

Operador	Qué hace	¿Se puede sobreescribir desde la CLI?
`=`	Asignación simple (expandida cuando se usa)	Sí (`make CC=clang`)
`:=`	Asignación inmediata (expandida en el momento)	Sí
`?=`	Condicional: solo asigna si **no está definida**	Solo si no viene de la CLI
`+=`	Concatena al final del valor actual	Sí

`=` vs `:=` : cuándo importa la diferencia

```
A = $(B)      # expande $(B) cuando se usa A (lazy)
B = hello
C := $(B)     # expande $(B) ahora mismo, aunque B cambie después

# Si B cambia después...
B = world

all:
[TAB]@echo "A = $(A)"   # A = world   (expandió B cuando se usó A)
[TAB]@echo "C = $(C)"   # C = hello  (ya estaba fijada con :=)
```

En la práctica, `:=` **es más seguro** para variables que dependen de otras: sabés exactamente qué valor van a tener. Pero `=` es más flexible cuando querés que las variables se resuelvan al final, después de que todas fueron definidas.

`?=` : el default configurable

```
CC ?= gcc     # Usá gcc por defecto, pero si el usuario pasó CC=clang, respetalo
CFLAGS ?= -Wall -Wextra   # mismo principio
```

¿Por qué importa? Porque la CLI tiene prioridad sobre `?=` :

```
$ make CC=clang    # clang gana porque viene de la línea de comandos
$ make             # gcc es el default (no vino nada por CLI)
```

`+=` : construir listas

```
CFLAGS += -std=c11
CFLAGS += -pedantic
# CFLAGS = -Wall -Wextra -std=c11 -pedantic
```

Cada `+=` **agrega al final**. Es perfecto para ir construyendo una lista de banderas sin reescribir todas las anteriores.

4.2 Variables que ya vienen con `make` (y las que gcc espera)

Hay variables que `make` conoce por defecto, y otras que el **compilador** conoce por convención. Es bueno distinguir ambas:

Variables de `make`

Variable	Default	Descripción
`CC`	`gcc`	El compilador C
`CXX`	`g++`	El compilador C++
`AR`	`ar`	El archivador (para bibliotecas estáticas)
`RM`	`rm -f`	El comando de borrado (`clean` usa esto)

Variables que gcc espera (por convención)

Variable	Qué va adentro	Ejemplo
`CFLAGS`	Banderas del compilador C	`-Wall -Wextra -O2`
`CPPFLAGS`	Banderas del preprocesador	`-I./include -DDEBUG=1`
`LDFLAGS`	Banderas del linker	`-L./lib -rpath=/opt/lib`
`LDLIBS`	Librerías para el linker	`-lm -lpthread -lmylib`

> ***Fijate la diferencia entre `LDFLAGS` y `LDLIBS`.*** *`LDFLAGS` son opciones del linker (dónde buscar librerías, dónde buscar paths). `LDLIBS` son las librerías en sí (el `-l...`). Es una convención de GNU: separás la **ubicación** (`-L`) de los **nombres** (`-l`). Si mezclás ambas cosas en una sola variable, el Makefile se vuelve confuso.*

4.3 La receta del linker: el orden importa

Cuando enlazás un programa con bibliotecas, el **orden** de los argumentos del linker importa:

```
$ gcc main.o utils.o -lm -o programa    # correcto: bibliotecas AL FINAL
$ gcc -lm main.o utils.o -o programa    # a veces funciona, a veces no
```

La regla general es simple: **las bibliotecas van al final**. El linker procesa los archivos de izquierda a derecha: si una biblioteca aparece antes que el código que la necesita, no la encuentra.

La receta completa del linker

En un Makefile profesional, la regla de enlace se ve así:

```
$(TARGET): $(OBJS)
[TAB]$(CC) $^ -o $@ $(LDFLAGS) $(LDLIBS)
```

Desglosado:

Parte	Qué es	Ejemplo de contenido
`$(CC)`	El compilador (gcc, clang, etc.)	`gcc`
`$^`	Todos los objetos	`main.o utils.o parser.o`
`-o $@`	Nombre del ejecutable.	
`$^`	Todos los archivos objeto y dependencias	`main.o utils.o parser.o libfoo.a`
`$(LDFLAGS)`	Opciones del linker	`-L./lib -rpath=/opt/lib`
`$(LDLIBS)`	Librerías para el linker	`-lm -lpthread -lmylib`

> ***La receta profesional completa.*** *Un Makefile de producción se parece a esto:*

```
CC = gcc
CFLAGS := -Wall -Wextra -std=c11 -O2 -g
CPPFLAGS += -I./include -DVERSION="\"1.0.0\""
LDFLAGS += -L./lib
LDLIBS += -lm -lpthread

TARGET = miapp
SRCS := $(wildcard src/*.c)
OBJS := $(patsubst src/%.c, build/%.o, $(SRCS))

.PHONY: all clean release debug

all: $(TARGET)

build/%.o: src/%.c | build
[TAB]$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@

$(TARGET): $(OBJS)
[TAB]$(CC) $^ -o $@ $(LDFLAGS) $(LDLIBS)

build:
[TAB]mkdir -p build

debug: CFLAGS += -fsanitize=address,undefined -DDEBUG
debug: clean all

release: CFLAGS += -O3 -flto
release: clean all

clean:
[TAB]$(RM) -r build $(TARGET)
```

Este Makefile incluye: `:=` para variables principales, `+=` para banderas adicionales, regla de patrón genérica que compila cualquier `.c` de `src/` a un `.o` en `build/`, order-only prerequisite (`| build`) para crear directorios, targets condicionales (`debug` y `release`), y sanitizers.

> ***Truco de laboratorio:*** `make -n` *o* `make --dry-run`. *Antes de ejecutar cualquier receta, ejecutá* `make -n`. *Imprime los comandos que SE EJECUTARÍA sin ejecutarlos. Úsalo antes de cambiar recetas.*

4.4 El Makefile ideal: resumen completo

```
CC ?= gcc
CFLAGS ?= -Wall -Wextra -std=c11
AR ?= ar
RM ?= rm -f

TARGET = programa
SRCS := $(wildcard src/*.c)
OBJS := $(patsubst src/%.c, build/%.o, $(SRCS))

CPPFLAGS += -I./include
LDFLAGS += -L./lib
LDLIBS += -lm

.PHONY: all clean install

all: $(TARGET)

build/%.o: src/%.c
[TAB]$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@

$(TARGET): $(OBJS)
[TAB]$(CC) $^ -o $@ $(LDFLAGS) $(LDLIBS)

clean:
[TAB]$(RM) -rf build/ $(TARGET)
```

Las claves: `?=` para CC/CFLAGS (sobreescribible), `:=` para TARGET/SRCS/OBJS, `+=` para banderas extra, regla de patrón genérica y `.PHONY`. Este esqueleto es tu punto de partida — en las siguientes partes añadimos bibliotecas estáticas, dinámicas y subdirectorios.

PARTE 5 — Bibliotecas estáticas

Una biblioteca estática (`.a` en Linux/macOS, `.lib` en Windows) es un **archivo ZIP** que contiene múltiples archivos objeto (`.o`) con un índice para que el linker encuentre cualquier símbolo. Al enlazar contra una biblioteca estática, el linker **copia** el código necesario en tu ejecutable.

5.1 ¿Qué es una biblioteca estática?

Si escribiste `mathops.c`, `vector.c` y `matrix.c` con funciones reutilizables, querés que otros programadores las usen **sin recompilar tu código cada vez**. La solución: empaquetar los `.o` en un `.a`:

```
gcc -c mathops.c vector.c matrix.c -I./include
ar rcs libmath.a mathops.o vector.o matrix.o
ar t libmath.a    # muestra: mathops.o  vector.o  matrix.o
```

La herramienta `ar` (archiver) usa tres opciones siempre juntas:

Opción	Qué hace
`r`	Reemplaza o agrega archivos (crea si no existe)
`c`	No genera mensaje si el archivo es nuevo
`s`	Escribe el índice (esencial para que el linker encuentre símbolos)

> ***Para guardar.*** *`ar rcs` es siempre las mismas cuatro letras. Sin `s`, el linker no encuentra símbolos.*

5.2 Cómo usar una biblioteca estática

```
gcc main.c -o programa -L./lib -lmath     # forma con -L y -l
gcc main.o -o programa ./lib/libmath.a    # equivalente directo
```

Opción	Qué significa	Ejemplo
`-L<dir>`	Agrega `<dir>` al path del linker	`-L./lib`
`-l<nombre>`	Busca `lib<nombre>.a` o `.so`	`-lmath` busca `libmath.a`

> ***La convención:*** *las bibliotecas se llaman siempre `libNOMBRE.a`. El linker busca `lib` + nombre + `.a` / `.so`.*

5.3 Biblioteca estática en el Makefile

```
STATIC_LIB = libmath.a
LIB_SRCS := mathops.c vector.c matrix.c
LIB_OBJS := $(patsubst %.c,build/%.o,$(LIB_SRCS))

.PHONY: all clean static

all: $(TARGET) $(STATIC_LIB)

$(STATIC_LIB): $(LIB_OBJS)
[TAB]$(AR) rcs $@ $^

build/%.o: src/%.c
[TAB]$(CC) $(CFLAGS) -c $< -o $@

clean:
[TAB]$(RM) -rf build/ $(TARGET) $(STATIC_LIB)
```

> ***Fíjate que*** `$(AR)` ***usa*** `$^`. *Esto ejecuta* `ar rcs libmath.a a.o b.o c.o` *en una línea.*

Práctica de la Parte 5

Ejercicio 21.5 — Crear una biblioteca estática

Conceptos · `ar`, `ar rcs`, bibliotecas estáticas, `-L`, `-l`

Consigna · Con archivos `mathops.c`, `vector.c`, `matrix.c`:

1. Regla de patrón para compilar `.c` → `.o` en `build/`.
2. Target `$(STATIC_LIB)` con `ar rcs $@ $^`.
3. Target `all` que construye ejecutable + biblioteca.
4. Target `clean` que borra todo.

Escenario de prueba

```
$ make clean && make
gcc -Wall -Wextra -c src/mathops.c -o build/mathops.o
gcc -Wall -Wextra -c src/vector.c -o build/vector.o
gcc -Wall -Wextra -c src/matrix.c -o build/matrix.o
ar rcs libmath.a build/mathops.o build/vector.o build/matrix.o

$ ar t libmath.a
build/mathops.o
build/vector.o
build/matrix.o
```

Identificadores sugeridos · `STATIC_LIB` , `LIB_SRCS` , `LIB_OBJS` **Criterios de aceptación**

- ☐ `libmath.a` se crea con `ar rcs` .
- ☐ `ar t libmath.a` muestra los tres objetos.
- ☐ Un programa externo enlaza con `-L. -lmath` .

PARTE 6 — Bibliotecas dinámicas

Una biblioteca dinámica (`.so` en Linux/macOS, `.dll` en Windows) se **carga en memoria** al ejecutar. El código no se copia en tu ejecutable: se comparte entre todos los procesos que lo usan.

6.1 ¿Qué es una biblioteca dinámica?

```
BIBLIOTECA ESTÁTICA (.a)          BIBLIOTECA DINÁMICA (.so)
Código copiado en el ejecutable   Código fuera, en un .so
Mayor tamaño del binario          Menor tamaño del binario
No necesita el .a al ejecutar    Necesita el .so al ejecutar
Un solo proceso por copia         Compartida entre procesos
```

6.2 Compilar una biblioteca dinámica

```
gcc -fPIC -c mathops.c vector.c matrix.c     # Paso 1: objetos position-independent
gcc -shared -o libmath.so mathops.o vec.o    # Paso 2: enlazar como .so
```

La bandera `-fPIC` (Position Independent Code) es **obligatoria** para `.so`. Sin ella, el linker falla con "relocation R_X86_64_32 against `...` can not be used when making a shared object".

6.3 Ejecutar con bibliotecas dinámicas

```
LD_LIBRARY_PATH=. ./miapp                      # Forma 1: variable de entorno
gcc main.c -o miapp -L. -lmath -Wl,-rpath,'$ORIGIN'  # Forma 2: rpath embebido
ldd miapp                                      # Ver dependencias
```

> ***$ORIGIN*** *= directorio donde vive el ejecutable. La forma más robusta de rutas relativas.*

6.4 Biblioteca dinámica en el Makefile

```
DYNAMIC_LIB = libmath.so
LIB_OBJS := $(patsubst %.c,build/%.o,$(LIB_SRCS))

$(DYNAMIC_LIB): $(LIB_OBJS)
[TAB]$(CC) -shared -o $@ $^

build/%.o: src/%.c
[TAB]$(CC) $(CFLAGS) -fPIC -c $< -o $@    # -fPIC obligatorio para .so
```

Sin `-fPIC` y `-shared`, no tenés una biblioteca dinámica.

Práctica de la Parte 6

Ejercicio 21.6 — Enlazar contra una biblioteca estática vs dinámica

Conceptos · `-L`, `-l`, `.a` vs `.so`, `ldd`, resolución de símbolos

Contexto · Ya sabés crear bibliotecas estáticas (`.a`) y ahora vas a entender cómo cambia el enlace cuando usás una biblioteca dinámica (`.so`). Este ejercicio te obliga a compilar y ejecutar el mismo programa contra ambas versiones.

Consigna · Tenés dos bibliotecas: `libmath.a` (estática) y `libmath.so` (dinámica). Escribí un Makefile con dos targets que generen el **mismo ejecutable** enlazado de forma distinta:

1. `./miapp-static` → enlazado contra `libmath.a`
2. `./miapp-dynamic` → enlazado contra `libmath.so` con rpath

Después, usá `ls -lh`, `ldd` y `time` para comparar ambos ejecutables.

Entrada · Un solo `main.c` que llama a funciones de `libmath`. **Salida** · Dos ejecutables con tamaños distintos, y un análisis comparativo.

Escenario de prueba

```
$ make static dynamic
gcc -Wall -Wextra -c main.c -o main.o
gcc main.o libmath.a -o miapp-static
gcc main.o -L. -lmath -Wl,-rpath,'$ORIGIN' -o miapp-dynamic

$ ls -lh miapp-*
-rwxr-xr-x 1 user user 20K miapp-static    # Más grande (código copiado)
-rwxr-xr-x 1 user user 16K miapp-dynamic   # Menor (enlace diferido)

$ ldd miapp-dynamic | grep math
libmath.so => ./libmath.so (0x...)

$ ldd miapp-static | grep math
(ninguno — no depende de nada)
```

Identificadores sugeridos · `STATIC_LIB`, `DYNAMIC_LIB`, `miapp-static`, `miapp-dynamic` **Restricciones/pistas**

- No escribas la librería ni el main: asumí que ya existen.
- El rpath es clave para que el ejecutable dinámico encuentre la `.so`.

Criterios de aceptación

- [ ] `miapp-static` se construye con `gcc main.o libmath.a -o miapp-static`.
- [ ] `miapp-dynamic` usa `-L. -lmath -Wl,-rpath,'$ORIGIN'`.
- [ ] `ldd miapp-static` no muestra dependencias de libmath.
- [ ] `ldd miapp-dynamic` muestra `libmath.so => ...`.

Ejercicio 21.7 — Crear una biblioteca dinámica

Conceptos · `-fPIC`, `-shared`, `.so`, `ldd`, `LD_LIBRARY_PATH`

Consigna · Con el proyecto del ejercicio 21.5, modificá el Makefile para:

1. Genere objetos con `-fPIC`.
2. Cree `libmath.so` con `-shared`.
3. El ejecutable enlaza contra la `.so` con rpath (`-Wl,-rpath,'$ORIGIN'`).

Escenario de prueba

```
$ make clean && make
gcc -Wall -Wextra -fPIC -c src/mathops.c -o build/mathops.o
gcc -shared -o libmath.so build/mathops.o ...
ldd miapp    # muestra: libmath.so => ./libmath.so (0x...)
```

Criterios de aceptación

- [ ] Se compila con `-fPIC`.
- [ ] `libmath.so` existe y `ldd` lo reporta como dependencia.
- [ ] El programa ejecuta sin error "cannot open shared object file".

Ejercicio 21.8 — Estática vs dinámica: medir el impacto

Consigna · Con la misma librería compilada como `.a` y como `.so`:

1. Medí tamaño de `libmath.a` vs `libmath.so`.
2. Medí tamaño del ejecutable enlazado contra cada una.
3. Cronometrá ambos programas.
4. Usá `strings`, `nm`, `ldd`.

Criterios de aceptación

- [ ] El ejecutable estático es más grande (código copiado).
- [ ] Se explicó el "linker selectivo" (copy-on-link).
- [ ] Ambos programas miden correctamente con `time` o similar.

PARTE 7 — Makefiles profesionales

Acá juntamos todo en un Makefile que se parece a los de proyectos reales: sanitizers, targets debug/release, dependencias automáticas y variables consultadas al compilador.

7.1 Sanitizers en desarrollo, optimización en producción

```
.PHONY: all clean debug release test coverage

debug: CFLAGS += -fsanitize=address,undefined -g -O0 -DDEBUG
debug: $(TARGET)

release: CFLAGS += -O3 -flto -DNDEBUG
release: clean all

test: debug
[TAB]./$^ --test    # asumiendo que el programa soporta --test

coverage: CFLAGS += --coverage -fprofile-arcs -ftest-coverage
coverage: $(TARGET)
[TAB]gcov *.c
```

Cada target redefine `CFLAGS` **solo para sí mismo** con `target: CFLAGS += ...`.

Variables consultadas al compilador

```
COMPILER_VERSION := $(shell $(CC) -dumpversion)

info:
[TAB]@echo "Compiler: $(CC) ($(COMPILER_VERSION))"
[TAB]@echo "CFLAGS: $(CFLAGS)"
[TAB]@echo "TARGET: $(TARGET)"
```

> ***Truco:*** `make info`. *Imprime toda la configuración del build. El primer target que todo Makefile serio debería tener.*

Generación automática de dependencias

Modificar un `.h` no recompila los `.c` que lo incluyen. Solución: generar reglas de dependencia automáticas:

```
gcc -MMD -MP -c main.c -o main.o   # genera main.d además de main.o
```

```
DEPS := $(OBJS:.o=.d)
-include $(DEPS)
```

`-MMD` genera el archivo `.d` con dependencias. `-MP` agrega reglas vacías para cabeceras (evita errores si borrás un header).

> ***Para guardar.*** `gcc -MMD -MP` *soluciona: "compilé un* `.c` *que depende de un* `.h`*, pero el* `.h` *cambió y make no lo supo".*

Ejercicio 21.9 — Makefile profesional completo

Consigna · Construí un Makefile con targets: `all`, `debug`, `release`, `clean`, `info`, `test`. Debe incluir bibliotecas estática Y dinámica, rpath, `-MMD -MP`, y sanitizers en debug.

Criterios de aceptación

- ☐ `make debug` compila con sanitizers activados.
- ☐ `make info` imprime toda la configuración del build.
- ☐ Los archivos `.d` se generan e incluyen con `-include $(DEPS)`.
- ☐ Bibliotecas estática y dinámica se crean correctamente.

Ejercicio 21.10 — Depuración de dependencias rotas

Consigna · Con el Makefile del ejercicio 21.9:

1. Compilá con `make debug`.
2. Borrá los `.d` generados (`rm build/*.d`).
3. Modificá un header incluido.
4. ¿Qué pasó al ejecutar `make`? ¿Por qué?
5. Restablecé y verificá que make detecta el cambio en `.h`.

Criterios de aceptación

- ☐ Se explicó el efecto de borrar los `.d`.
- ☐ Se explica `-MP` con ejemplo concreto.

PARTE 8 — Integrador final: proyecto multi-subdirectorio

Acá juntamos todo en un proyecto real: múltiples subdirectorios, bibliotecas estáticas, dinámicas, y un Makefile que construye todo desde cero con `make`.

8.1 Estructura del proyecto

```
mi-proyecto/
├── Makefile                    # Makefile raiz (orquesta todo)
├── include/                    # Headers publicos
│   ├── mathops.h
│   ├── vector.h
├── src/                        # Codigo fuente principal
│   ├── main.c
│   └── utils.c
├── lib/                        # Bibliotecas de la aplicacion
│   ├── Makefile                # Makefile para compilar librerias
│   ├── mathops.c
│   ├── vector.c
├── tests/                      # Pruebas unitarias
│   ├── Makefile
│   └── test_mathops.c
└── build/                      # Objetos compilados (generado)
```

Cada subdirectorio tiene su propio Makefile. El raiz orquesta todo con `$(MAKE) -C <dir>`.

8.2 Makefile de la biblioteca (lib/Makefile)

```
CC = gcc
CFLAGS := -Wall -Wextra -std=c11 -I../include

%.o: %.c
[TAB]$(CC) $(CFLAGS) -fPIC -c $< -o $@

libmath.a: mathops.o vector.o
[TAB]$(AR) rcs $@ $^

libmath.so: mathops.o vector.o
[TAB]$(CC) -shared -o $@ $^

clean:
[TAB]$(RM) -f *.o libmath.a libmath.so
```

8.3 Makefile raiz (Makefile)

```
CC ?= gcc
CFLAGS ?= -Wall -Wextra -std=c11 -O2 -g

TARGET = miapp
LIB_DIR = lib
BUILD_DIR = build

.PHONY: all clean debug release libs test

all: $(TARGET)

$(BUILD_DIR)/%.o: src/%.c
[TAB]@mkdir -p $(BUILD_DIR)
[TAB]$(CC) $(CFLAGS) -Iinclude -c $< -o $@

libs:
[TAB]$(MAKE) -C $(LIB_DIR)

$(TARGET): libs
[TAB]$(CC) $(BUILD_DIR)/*.o -o $@ -L$(LIB_DIR) -lmath -Wl,-rpath,'$ORIGIN'

debug: CFLAGS += -fsanitize=address,undefined -g -O0
debug: clean all

release: CFLAGS += -O3 -flto -DNDEBUG
release: clean all

test: libs
[TAB]$(MAKE) -C tests

info:
[TAB]@echo "CC = $(CC)"
[TAB]@echo "CFLAGS = $(CFLAGS)"
[TAB]@echo "TARGET = $(TARGET)"

clean:
[TAB]$(RM) -rf $(BUILD_DIR)/ $(TARGET)
[TAB]$(MAKE) -C $(LIB_DIR) clean || true
[TAB]$(MAKE) -C tests clean || true
```

`$(MAKE) -C <dir>` cambia al directorio y ejecuta `make`. Es la forma estándar de builds recursivos. El `-C` es una bandera de GNU make, no del compilador.

8.4 Makefile de tests (tests/Makefile)

```
CC = gcc
CFLAGS := -Wall -Wextra -std=c11 -g

TESTS = test_mathops

all: $(TESTS)

test_%: %.c ../lib/libmath.a
[TAB]$(CC) $(CFLAGS) $< -L../lib -lmath -o $@

clean:
[TAB]$(RM) -f $(TESTS)
```

PARTE 9 — Instalación, configuraciones y recursos

Ya construiste bibliotecas estáticas, dinámicas, targets de debug/release y dependencias automáticas. Pero un proyecto profesional **se instala** en el sistema del usuario: binarios en `/usr/local/bin`, bibliotecas en `/usr/local/lib`, headers en `/usr/local/include` y recursos (iconos, datos) en `/usr/local/share`. Además, suele ofrecer **opciones de compilación**: "¿querés soporte para Vulkan o SDL? Elegí al hacer `make`".

Esta parte cubre todo eso: cómo instalar, desinstalar, elegir backends desde la línea de comandos y distribuir recursos propios.

9.1 Variables de directorio (`PREFIX`, `BINDIR`, etc.)

La convención de Unix define variables estándar que todos los Makefiles serios respetan. Tu usuario las puede sobreescribir desde la línea de comandos:

```
PREFIX    ?= /usr/local
BINDIR    ?= $(PREFIX)/bin
LIBDIR    ?= $(PREFIX)/lib
INCLUDEDIR?= $(PREFIX)/include
DATADIR   ?= $(PREFIX)/share/miapp
MANPAGEDIR?= $(PREFIX)/share/man/man1

.PHONY: all install uninstall clean
```

El `?=` significa que el usuario puede cambiar cualquier directorio sin modificar el Makefile:

```
$ make PREFIX=/opt/miapp BINDIR=/opt/miapp/bin install
$ make PREFIX=$HOME/.local install    # instalación local, sin root
```

9.2 El target `install`

El target `install` copia los archivos a sus destinos. Usa `mkdir -p` para crear directorios y `\cp` (copia segura) o `install` (el comando del sistema):

```
install: $(TARGET)
[TAB]@mkdir -p $(BINDIR)
[TAB]@mkdir -p $(LIBDIR)
[TAB]@mkdir -p $(INCLUDEDIR)
[TAB]@mkdir -p $(DATADIR)
[TAB]$(INSTALL_BIN) $(TARGET) $(BINDIR)/$(TARGET)
[TAB]$(INSTALL_LIB) libmath.a $(LIBDIR)/libmath.a
[TAB]$(INSTALL_LIB) libmath.so $(LIBDIR)/libmath.so
[TAB]@install -m 644 include/*.h $(INCLUDEDIR)/
[TAB]@cp -r resources/* $(DATADIR)/
```

`$(INSTALL_BIN)`, `$(INSTALL_LIB)`. No uses `cp` a secas. GNU Make define por defecto: `INSTALL = cp`. Pero el target `install` del sistema (el comando `/usr/bin/install`) maneja permisos y tipos correctamente. Puedes usarlo así:

```
INSTALL_BIN ?= $(INSTALL) -D -m 755
INSTALL_LIB ?= $(INSTALL) -D -m 644
```

La bandera `-D` crea directorios padres si no existen. `-m 755` = ejecutable, `-m 644` = lectura universal.

9.3 El target `uninstall` y limpieza de recursos

Cada cosa que `install` crea, `uninstall` la elimina:

```
uninstall:
[TAB]$(RM) $(BINDIR)/$(TARGET)
[TAB]$(RM) $(LIBDIR)/libmath.a
[TAB]$(RM) $(LIBDIR)/libmath.so
[TAB]$(RM) -rf $(DATADIR)
[TAB]@echo "Uninstalled from $(PREFIX)"
```

Para guardar. *Si `install` hace algo (ej. actualiza cachés del sistema con `ldconfig`), `uninstall` debe revertirlo. El desinstalador es tan importante como el instalador.*

9.4 Opciones de compilación: feature toggles

Imaginate una aplicación que puede renderizar gráficos usando **múltiples backends**: SDL2, OpenGL o Vulkan. En vez de obligar al usuario a editar el Makefile a mano, le das opciones desde la CLI:

```
# Por defecto usa SDL2 si no se especifica nada
RENDERER ?= sdl

ifeq ($(RENDERER),sdl)
[TAB]CFLAGS += -DRENDERER_SDL -I/usr/include/SDL2
[TAB]LDLIBS += -lSDL2
[TAB]@echo ">> Building with SDL2 backend"
endif

ifeq ($(RENDERER),opengl)
[TAB]CFLAGS += -DRENDERER_OPENGL -I/usr/include
[TAB]LDLIBS += -lGL -lm
[TAB]@echo ">> Building with OpenGL backend"
endif

ifeq ($(RENDERER),vulkan)
[TAB]CFLAGS += -DRENDERER_VULKAN -I/usr/include/x86_64-linux-gnu
[TAB]LDLIBS += -lvulkan-1
[TAB]@echo ">> Building with Vulkan backend"
endif
```

El usuario elige al invocar `make`:

```
$ make                          # SDL2 (por defecto)
$ make RENDERER=opengl          # OpenGL
$ make RENDERER=vulkan clean    # Vulkan (con limpieza previa)
```

Más ejemplos de feature toggles

Esto no es solo para backends gráficos. Podés activar/desactivar:

Toggle	Qué habilita	Ejemplo de uso
`RENDERER=opengl`	Backend gráfico OpenGL	Aplicaciones visuales
`ENABLE_TESTS=yes`	Compila y corre tests	Desarrollo activo
`LTO=yes`	Link-time optimization	Producción máxima performance
`DEBUG_SYMBOLS=yes`	Símbolos de depuración	Debugging
`FEATURE_X=yes`	Una funcionalidad concreta	Plugins, módulos opcionales

```
# Ejemplo: habilitar LTO si el usuario lo pide
ifeq ($(LTO),yes)
[TAB]CFLAGS += -flto -ffat-lto-objects
[TAB]@echo ">> LTO enabled"
endif

# Ejemplo: tests compilados pero no ejecutados por defecto
ifeq ($(ENABLE_TESTS),yes)
[TAB]CFLAGS += -DTESTING_ENABLED
[TAB]OBJS += tests/main.o
[TAB]test-targets: $(TARGETS)
[TAB][TAB]$(MAKE) -C tests
endif
```

Headers condicionales

Las opciones de compilación se reflejan en el código con `#ifdef`:

```
#ifdef RENDERER_SDL
    renderer_init_sdl();
#elif defined(RENDERER_OPENGL)
    renderer_init_opengl();
#elif defined(RENDERER_VULKAN)
    renderer_init_vulkan();
#else
#error "No renderer selected. Use: make RENDERER=sdl|opengl|vulkan"
#endif
```

El `#error` es la forma en que el Makefile fuerza al usuario a elegir un backend si no especificó ninguno (cuando `RENDERER` viene vacío).

9.5 Recursos propios: iconos, datos, configuración

Una aplicación instalada suele necesitar archivos adicionales: íconos, temas, archivos de configuración, modelos 3D, texturas, sonidos, etc. Estos se copian a un directorio de datos durante `make install`:

```
DATADIR ?= $(PREFIX)/share/miapp

install:
[TAB]@mkdir -p $(DATADIR)
[TAB]@cp -r resources/* $(DATADIR)/
[TAB]@echo "Resources installed to $(DATADIR)"
```

Y en el programa, la ruta a los recursos se construye con una macro del preprocesador:

```
CPPFLAGS += -DRES_DIR="\"$(DATADIR)\""
```

```
const char *resource_path(const char *filename) {
    static char path[1024];
    snprintf(path, sizeof(path), "%s/%s", RES_DIR, filename);
    return path;
}
```

> ***Para guardar.*** *Si los recursos cambian después de la instalación, el usuario solo necesita volver a ejecutar `make install` (no `make` completo). El Makefile debe distinguir entre "compilar" y "instalar datos".*

9.6 El Makefile de producción: resumen final

```
# ── Variables configurables por el usuario ──
CC           ?= gcc
PREFIX       ?= /usr/local
RENDERER     ?= sdl
ENABLE_TESTS ?= no
LTO          ?= no

BINDIR   := $(PREFIX)/bin
LIBDIR   := $(PREFIX)/lib
INCLUDEDIR:= $(PREFIX)/include
DATADIR  := $(PREFIX)/share/miapp
MANPAGEDIR:= $(PREFIX)/share/man/man1

INSTALL_BIN  ?= $(INSTALL) -D -m 755
INSTALL_LIB  ?= $(INSTALL) -D -m 644

# ── Banderas del compilador ──
CFLAGS  := -Wall -Wextra -std=c11
CPPFLAGS += -I./include -DRES_DIR="\"$(DATADIR)\""

ifeq ($(RENDERER),sdl)
[TAB]CFLAGS += -DRENDERER_SDL -I/usr/include/SDL2
[TAB]LDLIBS += -lSDL2
[TAB]@echo ">> Backend: SDL2"
endif

ifeq ($(RENDERER),opengl)
[TAB]CFLAGS += -DRENDERER_OPENGL
[TAB]LDLIBS += -lGL -lm
[TAB]@echo ">> Backend: OpenGL"
endif

ifeq ($(RENDERER),vulkan)
[TAB]CFLAGS += -DRENDERER_VULKAN
[TAB]LDLIBS += -lvulkan-1
[TAB]@echo ">> Backend: Vulkan"
endif

ifeq ($(LTO),yes)
[TAB]CFLAGS += -flto -ffat-lto-objects
[TAB]@echo ">> LTO enabled"
endif

# ── Archivos ──
TARGET = renderer
SRCS   := $(wildcard src/*.c)
OBJS   := $(patsubst %.c,build/%.o,$(SRCS))
```

```
.PHONY: all install uninstall clean debug release test info

all: $(TARGET)

build/%.o: src/%.c
[TAB]@mkdir -p build
[TAB]$(CC) $(CFLAGS) -c $< -o $@

$(TARGET): $(OBJS)
[TAB]$(CC) $^ -o $@ $(LDFLAGS) $(LDLIBS)

install: all
[TAB]@mkdir -p $(BINDIR) $(DATADIR)
[TAB]$(INSTALL_BIN) $(TARGET) $(BINDIR)/$(TARGET)
[TAB]@cp -r resources/* $(DATADIR)/
[TAB]@echo "Installed to $(PREFIX)"

uninstall:
[TAB]$(RM) $(BINDIR)/$(TARGET)
[TAB]$(RM) -rf $(DATADIR)
[TAB]@echo "Uninstalled from $(PREFIX)"

debug: CFLAGS += -g -O0 -fsanitize=address,undefined
debug: clean all

release: CFLAGS += -O3 -DNDEBUG
release: clean all

test:
[TAB]$(MAKE) -C tests run

info:
[TAB]@echo "CC = $(CC)"
[TAB]@echo "PREFIX = $(PREFIX)"
[TAB]@echo "RENDERER = $(RENDERER)"
[TAB]@echo "CFLAGS = $(CFLAGS)"
[TAB]@echo "TARGET = $(TARGET)"

clean:
[TAB]$(RM) -rf build/ $(TARGET)
[TAB]$(MAKE) -C tests clean || true
```

Con este Makefile tenés un proyecto que el usuario puede:

1. **Configurar** (`make RENDERER=opengl LTO=yes PREFIX=$HOME/.local`)
2. **Compilar** en debug o release
3. **Instalar** con recursos y binarios en los directorios correctos
4. **Desinstalar** limpiamente

Práctica de la Parte 8

Ejercicio 21.11 — Proyecto multi-subdirectorio con bibliotecas anidadas

Conceptos · `$(MAKE) -C <dir>` , `.PHONY` , recursividad, `|| true` en limpieza

Contexto · Necesitás un proyecto con tres componentes independientes: una librería matemática (`lib/`), un programa principal (`src/`) y tests unitarios (`tests/`). Cada componente tiene su propio Makefile.

Consigna · Construí la estructura:

```
mi-proyecto/
├── Makefile          # Raíz (orquesta todo)
├── src/
│   ├── main.c
│   └── utils.c
├── lib/
│   ├── Makefile      # Compila libmath.a y libmath.so
│   ├── mathops.c
│   └── vector.c
├── include/
│   ├── mathops.h
│   └── vector.h
└── tests/
    ├── Makefile
    └── test_mathops.c
```

El Makefile raíz:

1. Construye la librería con `$(MAKE) -C lib`
2. Compila el programa principal vinculando contra la `.a` con rpath
3. Ofrece targets `debug` , `release` , `test` , `clean` que se propagan a subdirectorios

Escenario de prueba

```
$ make clean && make
make -C lib clean || true
gcc -Wall -Wextra -c src/main.c -o build/main.o
gcc build/main.o -Llib -lmath -Wl,-rpath,'$ORIGIN' -o miapp
./miapp
Procesando datos...

$ make test
make -C tests
gcc -Wall -Wextra -std=c11 -g test_mathops.c -L../lib -lmath -o test_mathops
./test_mathops
Running test_add: OK (0.00s)
```

Identificadores sugeridos · `miapp` , `LIB_DIR` , `BUILD_DIR` **Criterios de aceptación**

- [ ] Cada subdirectorio (`lib/` , `tests/`) tiene su Makefile independiente.
- [ ] El Makefile raíz usa `$(MAKE) -C <dir>` para orquestar.
- [ ] `make clean` en el raiz limpia todos los subdirectorios (con `|| true`).
- [ ] El programa ejecuta correctamente contra la librería local.

Práctica de la Parte 9

Ejercicio 21.12 — Makefile instalable con feature toggles (ejercicio jefe)

Conceptos · `PREFIX` , `BINDIR` , `DATADIR` , `make install/uninstall` , `ifeq` , `-D` , `#ifdef` , recursos, `INSTALL_BIN` , `INSTALL_LIB` , variables de directorio configurables

Contexto · Trabajás en un equipo que desarrolla una herramienta de línea de comandos llamada `vox` . No necesitás implementar la herramienta: su trabajo es crear un Makefile **completamente funcional** para construir e instalar el proyecto. Los archivos fuente (`.c` , `.h`) y los recursos (`.txt` , `.dat`) ya existen en la estructura del proyecto; tu objetivo es dominar `make` a nivel de distribución profesional.

La aplicación tiene tres "backends" posibles de procesamiento que se activan con una variable de make:

Variable	Valor	Qué hace
`ENGINE`	`ref`	Motor básico en software (por defecto)
`ENGINE`	`simd`	Usa instrucciones SIMD (SSE2/AVX2)
`ENGINE`	`gpu`	Usa OpenCL para aceleración por GPU

Además, el programa usa dos bibliotecas externas: `libmath.a` (estática) y `libspectrum.so` (dinámica). Al instalarse, copia un binario, un header público y recursos de datos a directorios configurables.

Consigna · Escribí **solo el Makefile**. No escribas los `.c`, ni las librerías, ni los recursos. Asumí que existen:

- `src/main.c` — incluye `vox.h` y usa `#ifdef ENGINE_REF` / `#ifdef ENGINE_SIMD` / `#ifdef ENGINE_GPU`
- `src/engine_ref.c`, `src/engine_simd.c`, `src/engine_gpu.c` — implementación de cada motor (condicional)
- `include/vox.h` — header público a instalar
- `lib/libmath.a` — biblioteca estática externa (ya compilada, en el proyecto)
- `lib/libspectrum.so` — biblioteca dinámica externa (ya compilada)
- `resources/config.txt`, `resources/defaults.dat` — archivos de datos

Tu Makefile debe:

1. **Variables configurables**: `PREFIX ?= /usr/local`, `ENGINE ?= ref`. El usuario debe poder hacer `make ENGINE=simdx86 PREFIX=$HOME/.local install`.

2. **Selección de backend**: usar `ifeq` para activar las banderas correctas:

```
ifeq ($(ENGINE),simd)
    CFLAGS += -DENGINE_SIMD -msse2 -mavx2
    LDLIBS += -lm
endif
ifeq ($(ENGINE),gpu)
    CFLAGS += -DENGINE_GPU
    LDLIBS += -lOpenCL
endif
# ENGINE=ref no necesita flags extra (código nativo)
```

3. **Compilación condicional**: compilar solo el engine seleccionado:

```
ifeq ($(ENGINE),simd)
    OBJS := src/main.o src/engine_simd.o
else ifeq ($(ENGINE),gpu)
    OBJS := src/main.o src/engine_gpu.o
else
    OBJS := src/main.o src/engine_ref.o
endif
```

4. **Enlazado**: contra `libmath.a` (estática, ruta directa) y `libspectrum.so` (dinámica, con rpath):

```
vox: $(OBJS) lib/libmath.a
    $(CC) $^ -o $@ -Llib -lspectrum -Wl,-rpath,'$ORIGIN/../lib' $(LDLIBS)
```

5. **Instalación**: con `PREFIX` configurable, copiando binario, header y recursos:

```
PREFIX    ?= /usr/local
BINDIR    := $(PREFIX)/bin
INCLUDEDIR:= $(PREFIX)/include/vox
LIBDIR    := $(PREFIX)/lib/vox
DATADIR   := $(PREFIX)/share/vox
MANPAGEDIR:= $(PREFIX)/share/man/man1

install: vox
    @mkdir -p $(BINDIR) $(INCLUDEDIR) $(LIBDIR) $(DATADIR) $(MANPAGEDIR)
    install -D -m 755 vox $(BINDIR)/vox
    install -D -m 644 include/vox.h $(INCLUDEDIR)/vox.h
    cp lib/libmath.a $(LIBDIR)/libmath.a
    cp lib/libspectrum.so $(LIBDIR)/libspectrum.so
    cp resources/* $(DATADIR)/
```

6. **Desinstalación**: revertir la instalación:

```
uninstall:
    $(RM) $(BINDIR)/vox
    $(RM) -rf $(INCLUDEDIR) $(LIBDIR) $(DATADIR) $(MANPAGEDIR)/vox.1*
    @echo "Uninstalled from $(PREFIX)"
```

7. **Target `info`**: imprimir toda la configuración activa:

```
info:
    @echo "CC = $(CC)"
    @echo "PREFIX = $(PREFIX)"
    @echo "ENGINE = $(ENGINE)"
    @echo "CFLAGS = $(CFLAGS)"
    @echo "OBJS = $(OBJS)"
```

8. **Debug / Release**: targets condicionales que redefinen `CFLAGS`:

```
debug: CFLAGS += -g -O0 -fsanitize=address,undefined
debug: clean vox

release: CFLAGS += -O3 -flto -DNDEBUG
release: clean vox
```

Escenario de prueba esperado (solo el Makefile se evalúa):

```
$ make ENGINE=simd info
>> Backend SIMD seleccionado
CC = gcc
PREFIX = /usr/local
ENGINE = simd
CFLAGS = -Wall -Wextra -std=c11 -DENGINE_SIMD -msse2 -mavx2
OBJS = src/main.o src/engine_simd.o

$ make ENGINE=simd PREFIX=$HOME/.local install
gcc ... -c src/main.c -o src/main.o
gcc ... -c src/engine_simd.c -o src/engine_simd.o
gcc src/main.o src/engine_simd.o lib/libmath.a -o vox \
    -Llib -lspectrum -Wl,-rpath,'$ORIGIN/../lib' -lm
install -D -m 755 vox $HOME/.local/bin/vox
install -D -m 644 include/vox.h $HOME/.local/include/vox/vox.h
cp lib/libmath.a $HOME/.local/lib/vox/
cp lib/libspectrum.so $HOME/.local/lib/vox/
cp resources/* $HOME/.local/share/vox/

$ make ENGINE=gpu uninstall
rm -f /usr/local/bin/vox
rm -rf /usr/local/include/vox /usr/local/lib/vox /usr/local/share/vox
/usr/local/share/man/man1/vox.1*
Uninstalled from /usr/local
```

Identificadores sugeridos · `Makefile`, `VOX`, `ENGINE`, `OBJS` **Restricciones**

- **Solo escribís el Makefile.** No implementes los `.c`, ni compiles librerías externas, ni crees recursos.
- El evaluador revisará que las reglas sean correctas y que todas las variables sean sobreescribibles desde la CLI.
- Usá `$@`, `$^`, `$<` donde aplique — no escribas nombres de archivo a mano en las recetas.

Criterios de aceptación

- ☐ `PREFIX ?= /usr/local` y se respeta en `install`.
- ☐ `ENGINE ?= ref` selecciona el engine correcto con `ifeq`.
- ☐ Se compila solo el `.c` del engine seleccionado (no los tres).
- ☐ El enlace usa `libmath.a` directo + `libspectrum.so` con rpath.
- ☐ `make install` copia binario, header y recursos a los directorios correctos.
- ☐ `make uninstall` elimina todo lo que `install` creó.
- ☐ `make info` imprime configuración activa completa.
- ☐ `make debug` y `make release` redefinen `CFLAGS` correctamente.
- ☐ El usuario puede hacer `make ENGINE=gpu PREFIX=$HOME/.local install`.

Ejercicio 21.13 — Hacer que un header roto no rompa todo

Conceptos · `-MMD`, `-MP`, dependencias implícitas, `#include` indirectos, `make clean -a`

Contexto · Tu Makefile tiene `-MMD -MP` para generar dependencias automáticas. Pero alguien borra un archivo `.h` que fue generado por un script externo y no se regeneró. Hacés `make`, el compilador falla, y ahora tenés que limpiar manualmente.

Consigna · Con un Makefile que tenga `-MMD -MP`:

1. Creá un archivo `generated.h` simulado (un `echo "int fake = 42;" > generated.h`).
2. Compilá un `main.c` que incluya `<generated.h>`.
3. Borrá `generated.h` y ejecutá `make` otra vez — ¿qué pasa?
4. Usá `-MP` para agregar una regla vacía a `generated.h.d` (si existe).
5. Intentá `make generated.h` como target phony que regera el archivo.

Criterios de aceptación

- [ ] Se explica por qué `make` falla al borrar un header dependiente.
- [ ] Se muestra cómo agregar `generated.h:` como target en el Makefile para regenerarlo.
- [ ] El Makefile permite `make generated.h && make` sin errores.

> ***Has llegado hasta aquí.*** *El capítulo más largo del libro. Aprendiste a pasar de "compilar con gcc" a tener un sistema de build profesional con bibliotecas estáticas, dinámicas, sanitizers, tests, instalación configurada y opciones de compilación. En el próximo capítulo (`22`), vamos a ver cómo hacer esto aún mejor con **CMake**.*