

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

Programas en varios archivos

Encabezados, `make`, tipos opacos y pruebas por la interfaz

Gerson Garrido

Semana 6

Hoy

1. Abstracción: interfaz e implementación
2. Compilar por separado y enlazar
3. Los errores de trabajar con varios archivos
4. Guardas de inclusión y `static`
5. `make`: qué se recompila y por qué
6. Tipos opacos y pruebas por la interfaz

> **Lectura**
>
> Apuntes de la semana 6. Código en `code/week-06`. Cada terminal de hoy empieza en esa carpeta.

SECCIÓN 01

Abstracción

01

Tres deudas pendientes

- **Semana 2:** “tomen como receta” estas líneas en las pruebas.

```
#define main trips_main  #include "trips.c"
```

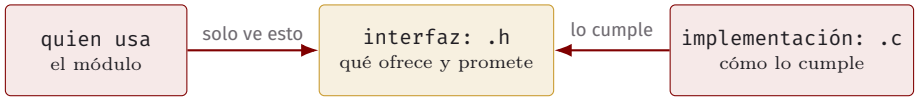
- **Semana 3:** `static` delante de una función, “se explica en la semana 6”.
- **Semana 3:** las líneas `#ifndef` de `image.h`, también.

La pregunta de hoy

¿Qué necesita saber cada parte de un programa sobre las demás? **Lo menos posible.**

Saber qué, sin saber cómo

Usan `printf` sabiendo **qué** hace. No saben **cómo**. Eso es abstracción.



- **Para leer:** basta el encabezado, mucho más corto que el código.
- **Para cambiar:** la implementación se reescribe sin tocar lo demás.

Una interfaz es una promesa

Lo que se puede ver, alguien lo va a usar, y ya no se podrá cambiar. Se muestra lo menos posible.

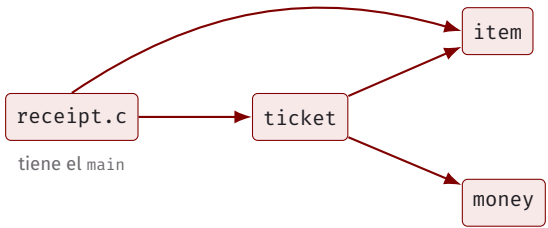
SECCIÓN 02

Compilar y enlazar

02



El ejemplo: un ticket de compra



`item`: un artículo y su costo. `money`: montos en céntimos. Cada módulo es un `.h` y un `.c`. Una flecha significa “incluye su encabezado”. `receipt.c` no sabe que existe `money`.

Un módulo: money

receipt/money.h

```h
#ifndef MONEY_H
#define MONEY_H

int money_discount(int cents, int percent);

void money_print(int cents);

#endif
```

Solo **prototipos**, los de Introducción a la Computación. Con eso, el compilador ya puede revisar cada llamada.

receipt/money.c

```c
#include "money.h"

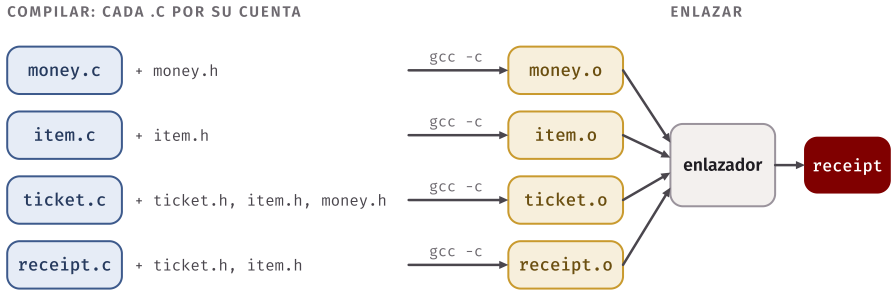
#include <stdio.h>

#define CENTS_PER_SOL 100

static int percent_of(int cents, int percent) {
    return cents * percent / 100;
}

int money_discount(int cents, int percent) {
    return cents - percent_of(cents, percent);
}
```

Dos etapas: compilar y enlazar



Un .h no se le pasa a gcc: entra porque el .c lo incluye.

- **Compilar:** cada `.c` por su cuenta. Mientras compila `receipt.c`, `gcc` no abre `ticket.c`.
- **Enlazar:** junta los objetos. Resuelve "aquí hace falta `ticket_print`".

A mano, una vez

terminal

```
$ cd receipt
$ make clean
rm -rf build
$ mkdir build
$ gcc -Wall -Wextra -std=c17 -c money.c -o build/money.o
$ gcc -Wall -Wextra -std=c17 -c item.c -o build/item.o
$ gcc -Wall -Wextra -std=c17 -c ticket.c -o build/ticket.o
$ gcc -Wall -Wextra -std=c17 -c receipt.c -o build/receipt.o
$ gcc build/receipt.o build/ticket.o build/item.o build/money.o -o build/receipt
$ build/receipt
Arroz      2  S/ 9.00
Aceite     1  S/ 12.90
Huevos    12  S/ 7.20
Total:         S/ 29.10
Con 10%:       S/ 26.19
```

La opción `-c` significa “solo compila, no enlaces”. Produce un **archivo objeto**.

Las notas que deja el compilador

terminal

```
$ cd receipt
$ nm build/receipt.o
0000000000000000 T main
                 U ticket_print
$ nm build/money.o
000000000000003c T money_discount
000000000000006c T money_print
0000000000000000 t percent_of
                 U printf
```

Las direcciones cambian según la máquina.

`T`	Lo define, y lo ofrece
`t`	Lo define, y es privado
`U`	Lo usa, y otro lo define

El enlazador junta cada `U` con un `T`.

`printf` lo trae la biblioteca de C.

`#include` copia texto

terminal

```
$ gcc -E -P receipt/receipt.c
struct item {
    const char *name;
    int quantity;
    int unit_cents;
};
int item_total(const struct item *item);
int items_total(const struct item items[], int n);
void ticket_print(const struct item items[], int n, int percent);
int main(void) {
```

`-E` se detiene después del preprocesador, y `-P` quita unas anotaciones. Esto es lo que ve el compilador: un solo texto, sin encabezados. Con esta idea se explica casi todo lo que sigue.

SECCIÓN 03

Errores

03



¿De qué etapa viene el error?

Causa	Mensaje	Etapa
Olvidar el prototipo	`implicit declaration`	Compilador
Tipos distintos en `.h` y `.c`	`conflicting types`	Compilador
Encabezado repetido, sin guarda	`redefinition of`	Compilador
Falta un archivo al enlazar	`undefined reference`	Enlazador
La misma función en dos `.c`	`multiple definition`	Enlazador
Un cuerpo en un `.h`	`multiple definition`	Enlazador

Cómo reconocerlo

Con un `.c` y un número de línea, es del compilador. Con `/usr/bin/ld`, es del enlazador, y cada archivo por separado está bien.

Olvidar el prototipo, olvidar un módulo

terminal

```
$ gcc -Wall -Wextra -std=c17 -c errors/no_prototype.c
errors/no_prototype.c: In function ‘main’:
errors/no_prototype.c:3:5: error: implicit declaration of function ‘money_print’ [-Wimplicit-function-declaration]
    3 |     money_print(1250);
      |     ^~~~~~~~~~
```

El compilador solo ve este archivo y lo que incluye. Falta `#include "money.h"`.

terminal

```
$ cd receipt
$ gcc receipt.c item.c money.c -o build/incomplete
receipt.c:(.text+0x40): undefined reference to ‘ticket_print’
collect2: error: ld returned 1 exit status
```

Quedó un `U` sin su `T`. El código está bien: falta enlazar `ticket.c`.

La misma función, dos veces

terminal

```
$ cd receipt
$ gcc -c ../errors/money_twin.c -o build/money_twin.o
$ gcc build/receipt.o build/ticket.o build/item.o build/money.o build/money_twin.o -o build/twin
money_twin.c:(.text+0x0): multiple definition of ‘money_print’; build/money.o:money.c:(.text+0x6c): first defined here
```

Dos `T` con el mismo nombre. Por eso los nombres llevan su módulo delante: `money_print`.

terminal

```
$ cd errors/header_body
$ gcc first.c second.c -o program
second.c:(.text+0x0): multiple definition of ‘twice’; /tmp/ccN6PXnB.o:first.c:(.text+0x0): first defined here
```

`twice.h` trae el cuerpo de la función. Como `#include` copia texto, quedó en los dos `.c`.

Qué va en cada archivo

En el `.h`	En el `.c`
Prototipos de lo que se ofrece	Cuerpos de las funciones
`struct` que quien usa debe conocer	Funciones `static` y `struct` privados
`#define` necesarios para usarlo	`#define` que solo usa ese archivo
Comentarios: qué se promete	Comentarios: cómo se hace

Un `struct` en un `.h` sí se puede repetir entre archivos: no genera código, es una descripción que se usa al compilar.

Cada `.c` incluye su propio encabezado

terminal

```
$ gcc -Wall -Wextra -std=c17 -c errors/drift.c
errors/drift.c:5:8: error: conflicting types for ‘area’; have ‘double(int,  int)’
    5 | double area(int width, int height) {
      |        ^~~~
In file included from errors/drift.c:3:
errors/drift.h:5:5: note: previous declaration of ‘area’ with type ‘int(int,  int)’
    5 | int area(int width, int height);
      |     ^~~~
```

Sin ese `#include`, compilaría sin queja, y los demás seguirían usando el prototipo viejo.

Idea clave

El enlazador solo mira nombres. Los tipos los comprueba el compilador, y solo con lo que cada archivo incluye.

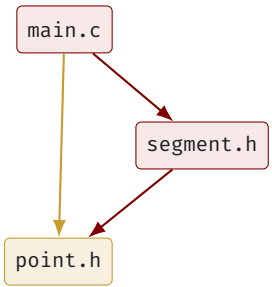
SECCIÓN 04

Guardas y static

04



El mismo encabezado, dos veces



`main.c` usa los dos tipos, y hace bien en incluir los dos. No tiene por qué saber qué incluye `segment.h`.

terminal

```
$ gcc -E -P errors/no_guard/main.c
struct point {
    int x;
    int y;
};
int point_sum(struct point p);
struct point {
    int x;
    int y;
};
```

Sin protección, `struct point` queda definido dos veces, y C no lo permite aunque sean iguales.

El error, y lo que hay que leer

terminal

```
$ gcc -Wall -Wextra -std=c17 -c errors/no_guard/main.c
In file included from errors/no_guard/segment.h:2,
                 from errors/no_guard/main.c:3:
errors/no_guard/point.h:2:8: error: redefinition of ‘struct point’
    2 | struct point {
      |        ^~~~~
In file included from errors/no_guard/main.c:2:
errors/no_guard/point.h:2:8: note: originally defined here
    2 | struct point {
      |        ^~~~~
```

Las primeras líneas dicen por qué camino llegó el encabezado. Lean el primer error: los que siguen son consecuencia.

La guarda de inclusión, paso a paso

receipt/item.h

```
1  #ifndef ITEM_H
2  #define ITEM_H
3  ...
4  #endif
```

"Si `ITEM_H` no está definido, sigue. Si no, sáltate todo hasta `#endif`."

Al procesar `receipt.c`	¿Definido?	Qué pasa
`item.h`, primera vez	No	Entra y lo define
`ticket.h`	Sí	Empieza a copiarlo
`item.h`, desde `ticket.h`	Sí	No copia nada

Por eso `struct item` salió una sola vez en `gcc -E`.

Error común: la guarda no arregla un `multiple definition`. Cada `.c` se compila por su lado, y recibe su copia.

`static` delante de una función

En la semana 3 cambiaba el tiempo de vida de una local. Aquí es otra cosa: solo este archivo puede usar la función.

terminal

```
$ cd receipt
$ gcc ../errors/call_private.c money.c -o build/private
call_private.c:(.text+0x10): undefined reference to ‘percent_of’
```

`call_private.c` escribe un prototipo correcto, y no le sirve: en `money.o` era una `t` minúscula.

- **Se puede cambiar:** para renombrarla, basta con mirar `money.c`.
- **No choca:** dos archivos pueden tener su propio `print_line`.
- **Comunica:** quien lee sabe que es un detalle.

Regla: toda función de un módulo que no está en su encabezado lleva `static`.

SECCIÓN 05

make

05



Una regla de `make`

receipt/Makefile

```
1  $(BUILD)/money.o: money.c money.h | $(BUILD)
2      $(CC) $(CFLAGS) -c $< -o $@
```

`$(BUILD)/money.o`	**Objetivo:** lo que se fabrica
`money.c money.h`	**Dependencias:** si alguna es más nueva, se vuelve a fabricar
`| $(BUILD)`	La carpeta debe existir, pero no importa su hora
`$(CC) ...`	**Receta:** empieza con un tabulador
`$<`, `$@`, `$^`	La primera dependencia, el objetivo, todas las dependencias

`make` no entiende C: compara horas de archivos. Las dependencias las escriben ustedes, y `make` les cree.

Desde cero, y otra vez

terminal

```
$ cd receipt
$ make clean
rm -rf build
$ make
mkdir -p build
gcc -Wall -Wextra -std=c17 -c receipt.c -o build/receipt.o
gcc -Wall -Wextra -std=c17 -c ticket.c -o build/ticket.o
gcc -Wall -Wextra -std=c17 -c item.c -o build/item.o
gcc -Wall -Wextra -std=c17 -c money.c -o build/money.o
gcc build/receipt.o build/ticket.o build/item.o build/money.o -o build/receipt
$ make
make: 'build/receipt' is up to date.
```

Sin argumentos, `make` fabrica el **primer objetivo** del archivo. Antes revisa sus dependencias, y las de ellas.

Qué se recompila

`touch` cambia la hora de un archivo, como si lo editaran. **Predigan antes de pasar.**

terminal

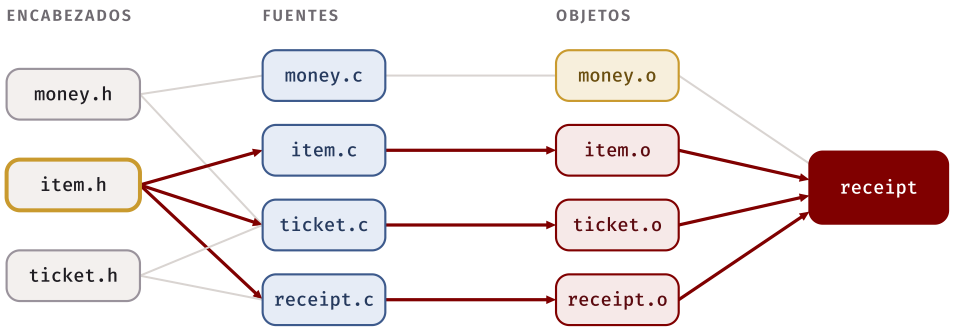
```
$ cd receipt
$ touch money.c
$ make
gcc -Wall -Wextra -std=c17 -c money.c -o build/money.o
gcc build/receipt.o build/ticket.o build/item.o build/money.o -o build/receipt
```

terminal

```
$ cd receipt
$ touch money.h
$ make
gcc -Wall -Wextra -std=c17 -c ticket.c -o build/ticket.o
gcc -Wall -Wextra -std=c17 -c money.c -o build/money.o
gcc build/receipt.o build/ticket.o build/item.o build/money.o -o build/receipt
```

Si cambia un `.h`: todo `.c` que lo incluya, directa o indirectamente.

Qué se recompila, siguiendo las líneas




```
$ touch item.h
```

item.h entra en item.c y receipt.c, y en ticket.c a través de ticket.h.

Un `Makefile` que miente

En `stale`, cada `.o` dice depender solo de su `.c`. Falta el encabezado.

terminal

```
$ cd stale
$ make clean
rm -rf build
$ make
mkdir -p build
gcc -Wall -Wextra -std=c17 -c main.c -o build/main.o
gcc -Wall -Wextra -std=c17 -c rectangle.c -o build/rectangle.o
gcc build/main.o build/rectangle.o -o build/show
$ build/show
3 x 5
$ cp rectangle_v2.h rectangle.h
$ make
make: 'build/show' is up to date.
```

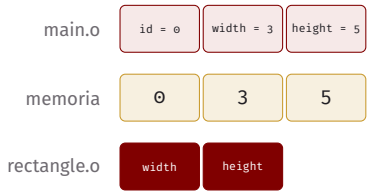
`rectangle_v2.h` agrega un campo `id` al inicio del `struct`. Y `make` dice que todo está al día.

Dos planos para una misma memoria

terminal

```
$ cd stale
$ touch main.c
$ make
gcc -Wall -Wextra -std=c17 -c main.c -o build/main.o
gcc build/main.o build/rectangle.o -o build/show
$ build/show
0 x 3
```

Sin advertencias. No hay ningún bug en `main.c` ni en `rectangle.c`.



`rectangle.o` sigue con el plano viejo: por eso escribe `0 x 3`.

Si algo imposible pasa después de cambiar un `.h`, prueben `make clean` y `make`. Si se arregla, al `Makefile` le falta una dependencia.

Dos ayudas

Para no adivinar las dependencias:

terminal

```
$ cd receipt
$ gcc -MM ticket.c
ticket.o: ticket.c ticket.h item.h money.h
```

El error de todo principiante: espacios en lugar del tabulador.

terminal

```
$ make -f errors/no_tab.mk
errors/no_tab.mk:2: *** missing separator.  Stop.
```

SECCIÓN 06

Documentar

06

El encabezado es parte de la interfaz

Por cada función, los comentarios contestan tres preguntas:

Qué promete	“Devuelve 1 si pudo y 0 si no hay memoria; en ese caso, no cambia nada”
Qué exige	“Exige: `tally_count(tally) > 0`”
Quién es owner	“Owner: quien llama, que debe liberarlo con `tally_free`”

- Si quien llama no cumple lo que se exige, la función no debe nada.
- Un comentario que menciona un campo o un `realloc` hace pública una decisión privada.
- `// Devuelve el conteo` sobre `tally_count` no agrega nada. Digan qué pasa con cero valores, con NULL, o si hay que liberar.

SECCIÓN 07

Tipos opacos

07

Mostrar el tipo, sin mostrar sus campos

tally/tally.h

```c
typedef struct tally Tally;

Tally *tally_new(void);
void tally_free(Tally *tally);
int tally_add(Tally *tally, int value);
int tally_count(const Tally *tally);
int tally_sum(const Tally *tally);
int tally_max(const Tally *tally);
```

tally/tally_summary.c

```c
struct tally {
    int count;
    int sum;
    int max;
};
```

`struct tally` anuncia que existe, sin decir qué tiene.

`typedef` le da otro nombre al tipo. La próxima semana lo vemos con calma.

Tipo opaco: declarado en el encabezado, y sus campos solo los conoce su `.c`.

¿Qué se puede hacer con él afuera?

terminal

```
$ gcc -Wall -Wextra -std=c17 -Itally -c errors/peek_tally.c
errors/peek_tally.c: In function ‘main’:
errors/peek_tally.c:7:11: error: storage size of ‘local’ isn’t known
    7 |     Tally local;
      |           ^~~~~
errors/peek_tally.c:9:25: error: invalid use of incomplete typedef ‘Tally’ {aka ‘struct tally’}
    9 |     printf("%d\n", tally->count);
      |                         ^~
errors/peek_tally.c:10:28: error: invalid application of ‘sizeof’ to incomplete type ‘Tally’ {aka ‘struct tally’}
   10 |     printf("%zu\n", sizeof(Tally));
      |                            ^~~~~
```

La línea 8, `Tally *tally = tally_new();`, no da error. ¿Por qué esa sí?

Qué necesita saber el compilador

Código	Qué necesita saber	¿Lo sabe?
`Tally local;`	Cuántos bytes reservar en el marco	No
`sizeof(Tally)`	Lo mismo	No
`tally->count`	Dónde está ese campo	No
`Tally *tally;`	Cuántos bytes ocupa un puntero	Sí
`tally_add(tally, 3)`	El prototipo	Sí

Afuera solo se tienen punteros, y solo se hace lo que permite el módulo. Ya no es buena conducta: lo impone el compilador.

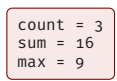
El costo: nadie afuera puede reservar un `Tally`. El módulo usa `malloc`, y quien llama a `tally_new` es el owner.

Dos representaciones, una interfaz

`tally_array.c`: guarda todos los valores



`tally_summary.c`: solo lleva el resumen



No guarda ningún valor.

Las dos cumplen `tally.h`. Quien usa el módulo no puede notar la diferencia.

La interfaz también limita. Si agregan `tally_median`, `tally_summary.c` ya no puede cumplirla: olvidó los valores. Se agrega a una interfaz con más cuidado que a una implementación.

Una prueba, dos enlaces

terminal

```
$ cd tally
$ make clean
rm -rf build
$ make
mkdir -p build
gcc -Wall -Wextra -std=c17 -c test_tally.c -o build/test_tally.o
gcc -Wall -Wextra -std=c17 -c tally_array.c -o build/tally_array.o
gcc build/test_tally.o build/tally_array.o -o build/test_array
gcc -Wall -Wextra -std=c17 -c tally_summary.c -o build/tally_summary.o
gcc build/test_tally.o build/tally_summary.o -o build/test_summary
$ build/test_array
Todas las pruebas pasaron
$ build/test_summary
Todas las pruebas pasaron
```

`test_tally.c` se compila una vez: el mismo código máquina corrió contra las dos. **Prueba por la interfaz:** solo usa lo que promete el encabezado, y sobrevive a un cambio de implementación.

Compárenlo con `stale`

terminal

```
$ cd tally
$ touch tally_summary.c
$ make
gcc -Wall -Wextra -std=c17 -c tally_summary.c -o build/tally_summary.o
gcc build/test_tally.o build/tally_summary.o -o build/test_summary
```

	`struct` en el `.h`	**`struct` en el `.c`**
Cambiar un campo	Todos los que lo incluyen	Se recompila ese `.c`
Quién conoce el plano	Todos	Solo el módulo
¿Puede pasar lo de `0 x 3`?	Sí, si falta una dependencia	No: nadie más usa los campos

Se acabó `#include "algo.c"`

- Incluir un `.c` copia su texto, también sus `static` y su `main`: por eso había que renombrarlo.
- La prueba quedaba pegada a la implementación, y nada comprobaba la interfaz.
- Era lo que se podía hacer con todo en un solo archivo.

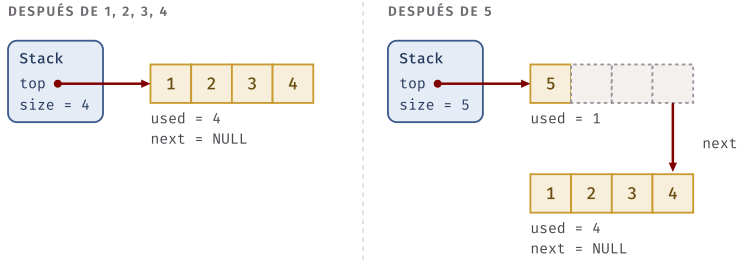
Desde esta semana

Una prueba incluye el encabezado, se compila aparte y se enlaza con los objetos del módulo. Nunca hace `#include` de un `.c`.

¿Y el `main` que estorbaba?

Va en un archivo propio y muy corto, como `receipt.c`. Lo demás pasa a un módulo, y ya se puede probar.

Bloques enlazados: otra forma de guardar



- `stack_push`: si el bloque de arriba está lleno, primero crea otro y lo pone encima.
- `stack_pop`: si el bloque de arriba queda vacío, lo libera, y `top` pasa al de abajo.
- `struct block *next` adentro de `struct block`: es un puntero, y su tamaño se conoce. La misma razón de los tipos opacos.
- Para liberar todo, guarden `next` antes de cada `free`.

El módulo de imágenes, ahora opaco

Hoy, en `filters.c`:

```
img->pixels[y * img->width + x]
```

Después de la conversión:

```
image_get(img, x, y)
image_set(img, x, y, value)
```

Mover el `struct` al `.c` y compilar: los errores son la lista de todo lo que dependía de la representación.

- **Hay que decidir qué se muestra.** `invert` necesita `image_max_value`.
- **Son un lugar donde revisar.** Con `x` igual a `width`, se cae en la fila siguiente, y ni ASan lo nota. Un `assert` en `image_get` sí.
- **Tiene un costo.** Cada píxel es una llamada a función.

SECCIÓN 08

Laboratorio

08

Laboratorio de esta semana

Sesión 1

1. Leer `market.c`, dividirlo en `sales`, `report` y `main`, y escribir su `Makefile`.
2. Escribir el módulo `Fraction`, opaco, con sus pruebas.

Sesión 2

3. `Stack` dos veces: arreglo que crece y bloques enlazados. Una sola prueba, ya hecha.
4. Convertir el módulo de imágenes a un tipo opaco.

Comandos

`make lab` y `make test-lab`. El `README.md` de `lab/` da los nombres exactos y marca lo que puede quedar para antes de la siguiente sesión.

Para la próxima clase

- Terminen el laboratorio: `make test-lab` debe pasar.
- Sepan responder sin mirar:
  1. ¿Qué error da definir la misma función en dos `.c`? ¿Y olvidar el prototipo?
  2. ¿Para qué sirve una guarda de inclusión?
  3. ¿Por qué un programa que usa un tipo opaco no puede declarar una variable de ese tipo, sino solo un puntero?
  4. Si cambian un `.h`, ¿qué archivos hay que recompilar?

Semana 7

Tipos: `typedef` con calma, `enum` y uniones etiquetadas. Elegir bien un tipo también evita bugs.