

Introducción a la Computación

Funciones y pruebas

Paso por valor, pila de llamadas, `assert` y gdb

Gerson Garrido

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Definir y llamar funciones. `void` y `return`
2. Paso por valor y alcance de las variables
3. La pila de llamadas, con un dibujo
4. Separar los cálculos de la entrada y salida
5. Probar con `assert`
6. Depurar con método, y `gdb`

> **Lectura**
>
> Apuntes de la semana 7. Código en `code/week-07`.

SECCIÓN 01

Definir y llamar

01

Para qué sirven las funciones

Ya usan funciones: `main`, `printf`, `scanf`, `sqrt`, `rand`. Hoy escribimos las nuestras.

Para qué sirven las funciones

Ya usan funciones: `main`, `printf`, `scanf`, `sqrt`, `rand`. Hoy escribimos las nuestras.

- **No repetir.** `adder4.c` tenía la misma cuenta cuatro veces. Una función se escribe una vez y se corrige en un solo lugar.

Para qué sirven las funciones

Ya usan funciones: `main`, `printf`, `scanf`, `sqrt`, `rand`. Hoy escribimos las nuestras.

- **No repetir.** `adder4.c` tenía la misma cuenta cuatro veces. Una función se escribe una vez y se corrige en un solo lugar.
- **Dar nombres.** `is_passing(grade)` dice lo que `grade >= 11` obliga a deducir.

Para qué sirven las funciones

Ya usan funciones: `main`, `printf`, `scanf`, `sqrt`, `rand`. Hoy escribimos las nuestras.

- **No repetir.** `adder4.c` tenía la misma cuenta cuatro veces. Una función se escribe una vez y se corrige en un solo lugar.
- **Dar nombres.** `is_passing(grade)` dice lo que `grade >= 11` obliga a deducir.
- **Probar por partes.** Una función pequeña se comprueba sola.

Para qué sirven las funciones

Ya usan funciones: `main`, `printf`, `scanf`, `sqrt`, `rand`. Hoy escribimos las nuestras.

- **No repetir.** `adder4.c` tenía la misma cuenta cuatro veces. Una función se escribe una vez y se corrige en un solo lugar.
- **Dar nombres.** `is_passing(grade)` dice lo que `grade >= 11` obliga a deducir.
- **Probar por partes.** Una función pequeña se comprueba sola.

> **Idea clave**
>
> Una buena función hace **una sola cosa**. Si para nombrarla necesitan la palabra “y”, probablemente son dos.

En pseudocódigo: `devolver`

pseudocódigo

```
algoritmo mayor_de_dos(a, b)
    si a > b entonces
        devolver a
    sino
        devolver b
    fin si
fin algoritmo
```

- Lo que está entre paréntesis es lo que recibe.
- `devolver` termina el algoritmo y le entrega un valor a quien lo pidió.
- Otro algoritmo lo usa como una operación:
  `m ← mayor_de_dos(a, b)`

Una función en C

square.c

```c
int square(int x) {
    int result = x * x;
    return result;
}
```

Definir no ejecuta: solo dice qué hacer cuando la llamen.

Pieza	Qué dice
`int`	Tipo de retorno: qué devuelve
`square`	Nombre, para llamarla
`int x`	Parámetro: recibe un valor de quien llama
`{ ... }`	Cuerpo: corre en cada llamada
`return`	Termina y entrega el valor

Qué pasa en una llamada

square.c

```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

Qué pasa en una llamada

square.c

```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

1. Se calcula el argumento. Si `n` vale 5, da 5.

Qué pasa en una llamada

square.c

```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

1. Se calcula el argumento. Si `n` vale 5, da 5.
2. La ejecución **salta** a `square`, y `x` recibe el 5.

Qué pasa en una llamada

square.c

```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

1. Se calcula el argumento. Si `n` vale 5, da 5.
2. La ejecución **salta** a `square`, y `x` recibe el 5.
3. Corre el cuerpo, y `return` entrega 25.

Qué pasa en una llamada

square.c

```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

1. Se calcula el argumento. Si `n` vale 5, da 5.
2. La ejecución **salta** a `square`, y `x` recibe el 5.
3. Corre el cuerpo, y `return` entrega 25.
4. Se **vuelve** al punto de la llamada. Ahí `square(n)` vale 25.

Qué pasa en una llamada

square.c

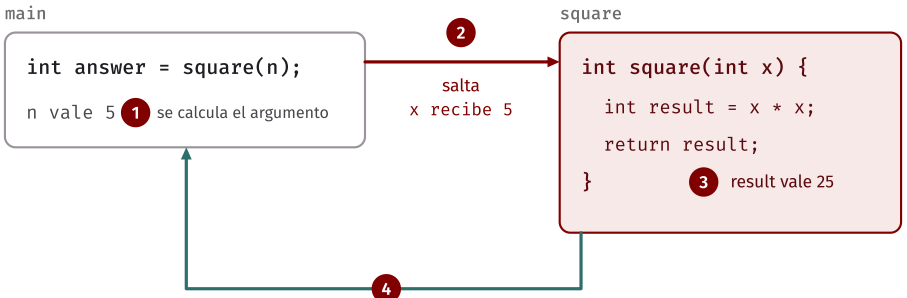
```c
int answer = square(n);
printf("%d\n", square(2) + square(n + 1));
```

terminal

```
$ echo 5 | build/square
5 al cuadrado es 25
40
```

1. Se calcula el argumento. Si `n` vale 5, da 5.
2. La ejecución **salta** a `square`, y `x` recibe el 5.
3. Corre el cuerpo, y `return` entrega 25.
4. Se **vuelve** al punto de la llamada. Ahí `square(n)` vale 25.
5. Una llamada es una expresión: la segunda línea es 4 + 36.

La llamada, dibujada



vuelve al punto de la llamada: ahí, square(n) vale 25, y se guarda en answer

Parámetro y argumento

Parámetro

La variable de la definición: `x` en `int square(int x)`.

Argumento *(argument)*

El valor de la llamada: el valor de `n` en `square(n)`, o el 2 en `square(2)`.

- Se asignan **por posición**: el primero al primero.
- Los nombres no importan. El argumento puede llamarse igual, o distinto, o no ser una variable.

Una función puede llamar a otra

max_of_three.c

```c
int max_of_two(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}

int max_of_three(int a, int b, int c) {
    return max_of_two(max_of_two(a, b), c);
}
```

Con `3 9 5`, lo de adentro va primero:

Llamada	Devuelve
`max_of_two(3, 9)`	9
`max_of_two(9, 5)`	9

No hace falta `else`: `return` ya terminó la función.

Lo que el compilador revisa y lo que no

Si falta un argumento, no compila:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/wrong_count.c -o build/wrong_count
errors/wrong_count.c: In function ‘main’:
errors/wrong_count.c:12:20: error: too few arguments to function ‘max_of_two’
   12 |     printf("%d\n", max_of_two(5));
      |                    ^~~~~~~~~
```

Si el tipo no coincide, C convierte sin preguntar. Aquí se pierde el decimal:

errors/double_argument.c

```c
1  printf("%d\n", square(2.5));
```

terminal

```
$ build/double_argument
4
```

SECCIÓN 02

void y return

02



Funciones `void` y sin parámetros

print_line.c

```c
void print_line(int width) {
    for (int i = 0; i < width; i++) {
        printf("-");
    }
    printf("\n");
}

void print_header(void) {
    print_line(20);
    printf("Boleta de notas\n");
    print_line(20);
}
```

- Tipo de retorno `void`: **no devuelve nada**. Se usa por lo que hace.
- Termina en la llave final, o antes con `return;`.
- `(void)` en los parámetros: **no recibe nada**.
- Se llama con paréntesis vacíos: `print_header();`
- Guardar su valor no compila: `int y = print_twice(4);` da `void value not ignored`.

Un resultado que nadie recibe

errors/ignored_result.c

```c
int a = 5;
square(a);
printf("%d\n", a);
```

terminal

```
$ build/ignored_result
5
```

Error común

`gcc` no avisa. `square` calcula 25 y lo devuelve, y se pierde. Lo correcto es `a = square(a);`.

Sin parámetros, y aun así útil

roll_die.c

```c
int roll_die(void) {
    return 1 + rand() % 6;
}
```

Cada llamada puede dar un valor distinto: `rand` avanza. Con `srand((unsigned) time(NULL))` en `main`, cada corrida cambia, salvo dentro del mismo segundo.

`(void)` y `()` no son lo mismo

Con `int get_year()`, `gcc` no revisa las llamadas: `get_year(1, 2, 3)` compila, y es comportamiento indefinido. Con `(void)`, da `too many arguments`. Siempre `(void)`.

Prototipos: anunciar antes de definir

prototype.c

```c
int cube(int x);

int main(void) {
    printf("%d\n", cube(3));
    return 0;
}

int cube(int x) {
    return x * x * x;
}
```

- `gcc` lee de arriba hacia abajo: al llamar, ya debe conocer la función.
- El **prototipo** dice tipo, nombre y parámetros, sin cuerpo.
- Sin prototipo, y con la definición abajo, no compila: `implicit declaration of function`.
- Otra solución: definir las funciones antes, y `main` al final.
- `stdio.h` trae el prototipo de `printf`.

Varios `return`: salir en cuanto se sabe

pseudocódigo

```
algoritmo tiene_cifra(n, cifra)
    mientras n > 0 hacer
        si n mod 10 = cifra entonces
            devolver 1
        fin si
        n ← n div 10
    fin mientras
    devolver 0
fin algoritmo
```

has_digit.c

```c
int has_digit(int n, int digit) {
    while (n > 0) {
        if (n % 10 == digit) {
            return 1;
        }
        n = n / 10;
    }
    return 0;
}
```

Responde una pregunta: 1 es sí, y 0 es no. Se usa como condición: `if (has_digit(n, digit))`. El primer `return` que se ejecuta termina la función.

“No está” solo se sabe al final

errors/has_digit_wrong.c

```
1  if (n % 10 == digit) {
2      return 1;
3  } else {
4      return 0;
5  }
```

terminal

```
$ echo 26 2 | build/has_digit_wrong
No aparece
```

Decidió con la primera cifra que miró, el 6. `gcc` no avisa.

Error común

El `return 0` va **después** del ciclo. Lo van a necesitar en `is_prime`.

Llegar al final sin `return`

errors/missing_return.c

```c
int is_passing(int grade) {
    if (grade >= 11) {
        return 1;
    }
}
```

- Con una nota de 5, ¿qué devuelve?
- Usar ese valor es **comportamiento indefinido**.
- No le crean a ninguna corrida. Solo `main` es excepción: llegar a su final equivale a `return 0`.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/missing_return.c -o build/missing_return
errors/missing_return.c: In function ‘is_passing’:
errors/missing_return.c:8:1: warning: control reaches end of non-void function [-Wreturn-type]
```

SECCIÓN 03

Paso por valor y alcance

03

Paso por valor: el parámetro es una copia

by_value.c

```c
void add_one(int x) {
    x = x + 1;
    printf("En add_one, x vale %d\n", x
      );
}
...
int x = 1;

add_one(x);
printf("En main, x vale %d\n", x);
```

terminal

```
$ build/by_value
En add_one, x vale 2
En main, x vale 1
```

Línea	x de main	x de add_one
`int x = 1;`	1	no existe

Paso por valor: el parámetro es una copia

by_value.c

```c
void add_one(int x) {
    x = x + 1;
    printf("En add_one, x vale %d\n", x
      );
}
...
int x = 1;

add_one(x);
printf("En main, x vale %d\n", x);
```

terminal

```
$ build/by_value
En add_one, x vale 2
En main, x vale 1
```

Línea	x de main	x de add_one
`int x = 1;`	1	no existe
se copia el 1	1	1

Paso por valor: el parámetro es una copia

by_value.c

```c
void add_one(int x) {
    x = x + 1;
    printf("En add_one, x vale %d\n", x
      );
}
...
int x = 1;

add_one(x);
printf("En main, x vale %d\n", x);
```

terminal

```
$ build/by_value
En add_one, x vale 2
En main, x vale 1
```

Línea	x de main	x de add_one
`int x = 1;`	1	no existe
se copia el 1	1	1
`x = x + 1;`	1	2

Paso por valor: el parámetro es una copia

by_value.c

```c
void add_one(int x) {
    x = x + 1;
    printf("En add_one, x vale %d\n", x
      );
}
...
int x = 1;

add_one(x);
printf("En main, x vale %d\n", x);
```

terminal

```
$ build/by_value
En add_one, x vale 2
En main, x vale 1
```

Línea	x de main	x de add_one
`int x = 1;`	1	no existe
se copia el 1	1	1
`x = x + 1;`	1	2
termina `add_one`	1	ya no existe

La copia, paso a paso

LA PILA

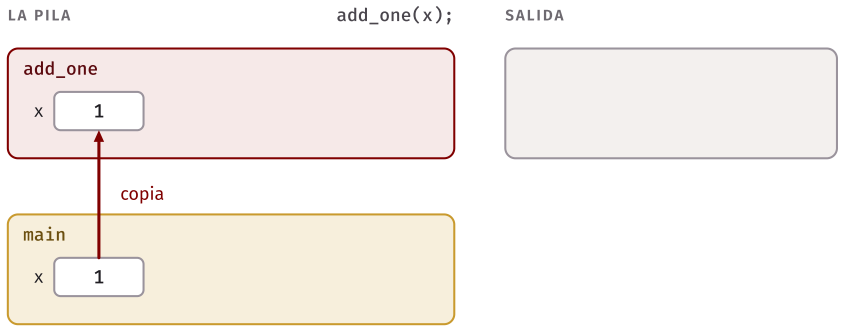
```
int x = 1;
```

SALIDA



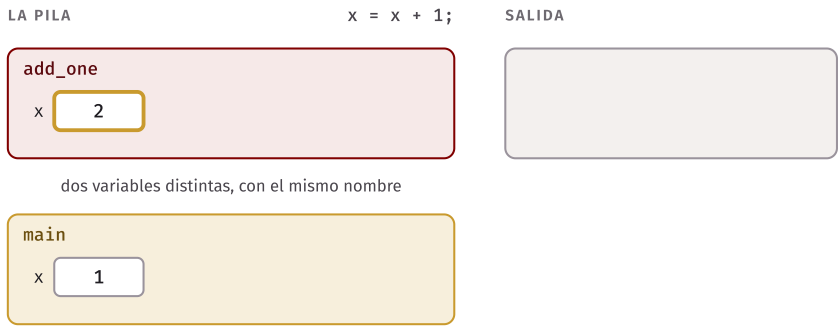
main crea su x, con el valor 1

La copia, paso a paso



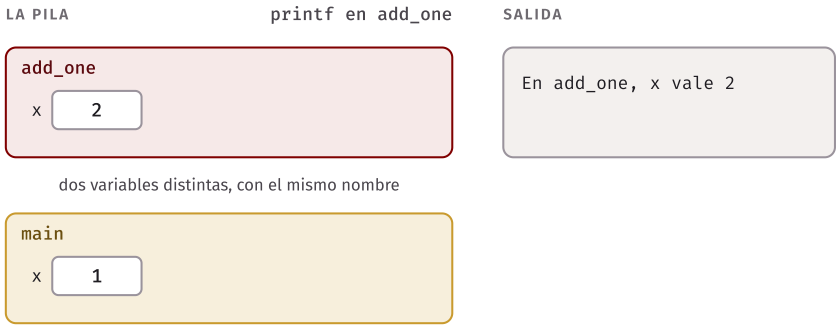
se llama a add_one: nace otro x y recibe una copia del 1

La copia, paso a paso



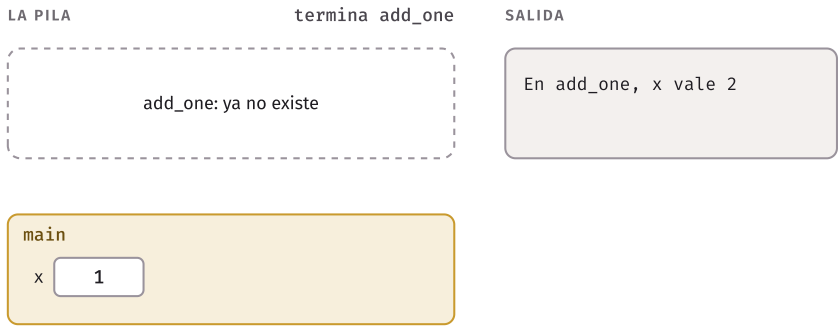
add_one cambia su copia; el x de main no se entera

La copia, paso a paso



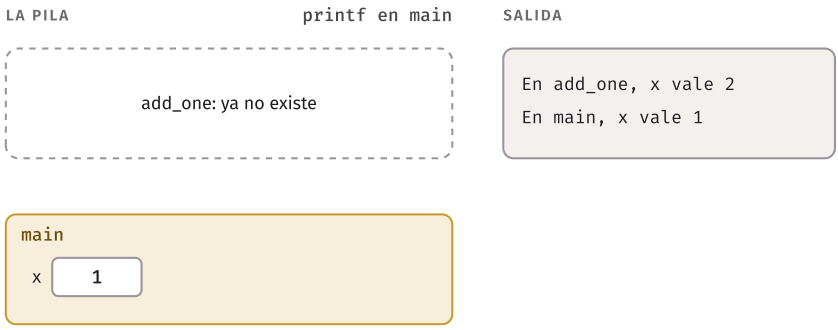
escribe el x que ve: el de add_one

La copia, paso a paso



su marco se retira y la copia deja de existir

La copia, paso a paso



el x de main sigue en 1

La salida es `return`

by_value.c

```c
int plus_one(int x) {
    return x + 1;
}
...
x = plus_one(x);
```

Idea clave

C pasa **siempre por valor**. La información entra por los parámetros y sale por `return`. Una función no cambia las variables de quien la llama a través de ellos.

Un intercambio que no intercambia

try_swap.c

```c
void try_swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf("En try_swap: a = %d, b =
     %d\n", a, b);
}
```

terminal

```
$ build/try_swap
En try_swap: a = 8, b = 3
En main:     a = 3, b = 8
```

- Intercambió las copias, y las copias desaparecieron.
- `return` no ayuda: una función devuelve **un solo valor**.
- Por eso `scanf("%d", &n)` recibe la **dirección** de `n`, no su valor. Volvemos a esa idea en la semana 10.

Alcance *(scope)*: dónde existe un nombre

scope.c

```c
int twice(int x) {
    int result = x * 2;
    return result;
}

int triple(int x) {
    int result = x * 3;
    return result;
}
```

- Una variable **local** se usa desde su declaración hasta que se cierra su bloque.
- Tres `result` en `scope.c`, contando el de `main`: no tienen nada que ver.
- Una función no ve las locales de otra: `‘result’ undeclared`.
- La `i` de `for (int i = 0; ...)` solo existe en ese `for`.
- **Sin variables globales**: pasen los datos por parámetros.

Una local nace y muere en cada llamada

count_calls.c

```c
int count_calls(void) {
    int calls = 0;
    calls = calls + 1;
    return calls;
}
```

terminal

```
$ build/count_calls
1
1
1
```

Intenta contar sus llamadas y no puede: cada llamada crea un `calls` nuevo. No recuerda la anterior.

SECCIÓN 04

La pila de llamadas

04



La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

```
main
total sin valor
```

Empieza el programa.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

`sum_of_squares`
`a` = 3, `b` = 4, `first` sin valor

`main`
`total` sin valor

`main` llama a `sum_of_squares(3, 4)`.

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

`square`
`x` = 3, `result` sin valor

`sum_of_squares`
`a` = 3, `b` = 4, `first` sin valor

`main`
`total` sin valor

Se llama a `square(3)`: se copia el valor de `a` en `x`.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

`square`
`x` = 3, `result` = 9

`sum_of_squares`
`a` = 3, `b` = 4, `first` sin valor

`main`
`total` sin valor

`square(3)` ya calculó `result`.

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

`sum_of_squares`
`a` = 3, `b` = 4, `first` = 9

`main`
`total` sin valor

Devolvió 9. Su marco se retiró.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

`square`: un marco nuevo
`x` = 4, otro `result`

`sum_of_squares`
`a` = 3, `b` = 4, `first` = 9

`main`
`total` sin valor

Se llama a `square(4)`.

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

int main(void) {
    int total = sum_of_squares(3, 4);
```

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

`sum_of_squares`
`a` = 3, `b` = 4, `first` = 9, `second` = 16

`main`
`total` sin valor

Devolvió 16.

La pila, paso a paso

call_chain.c

```c
int square(int x) {
    int result = x * x;
    return result;
}

int sum_of_squares(int a, int b) {
    int first = square(a);
    int second = square(b);
    return first + second;
}

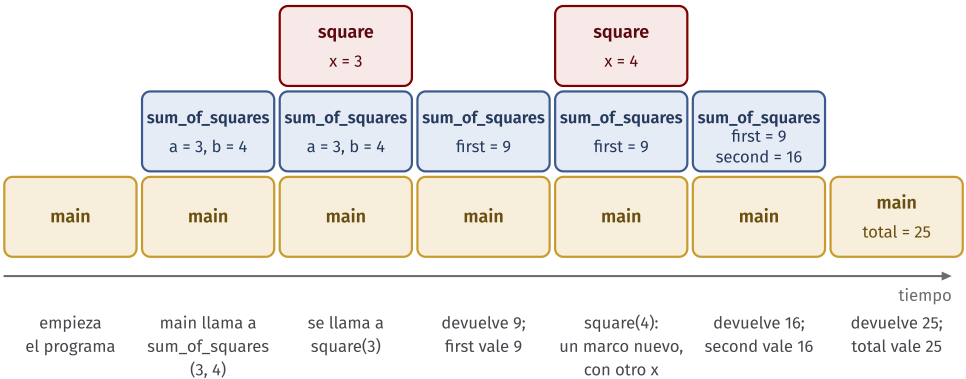
int main(void) {
    int total = sum_of_squares(3, 4);
```

Cada llamada pone un **marco** *(stack frame)* encima. Al terminar, se retira.

```
main
total = 25
```

Solo queda `main`.

La pila a lo largo del programa



Un modelo, no una promesa sobre la memoria

Lo que garantiza C

Cada llamada tiene sus propias variables, y al volver se sigue donde se quedó.

Lo que C no promete

En qué dirección queda una local, ni hacia dónde crece la pila. El compilador puede guardarla en otro lugar, o eliminarla.

Para qué sirve el dibujo

Para razonar: explica el paso por valor, el alcance y por qué se usa `return`. Hoy lo veremos con `backtrace` en un programa real.

SECCIÓN 05

Separar y probar

05

Separar los cálculos de la entrada y salida

discount_mixed.c: mal partido

```c
void discount_and_print(int total) {
    if (total >= 10000) {
        total = total - total / 10;
    }
    ...
}
```

discount.c: cada una, su trabajo

```c
int price_after_discount(int total) {
    if (total >= 10000) {
        return total - total / 10;
    }
    return total;
}
```

Calcula y escribe: no deja ver el resultado.

> **Idea clave**
>
> Calculen sin `scanf` ni `printf`. Así se puede probar.

Probar con `assert`

discount_tests.c

```c
#include <assert.h>
#include <stdio.h>
...
assert(price_after_discount(12050) == 10845);   // caso normal con descuento
assert(price_after_discount(8000) == 8000);     // caso normal sin descuento
assert(price_after_discount(9999) == 9999);     // último valor sin descuento
assert(price_after_discount(10000) == 9000);    // primer valor con descuento
assert(price_after_discount(0) == 0);           // el menor total válido

printf("Todas las pruebas pasaron\n");
```

`assert(condición);`

Si es verdadera, no hace nada. Si es falsa, escribe qué falló y **termina el programa**.

Una prueba que encuentra un bug

count_digits_bug.c

```c
int count_digits(int n) {
    int count = 0;
    while (n > 0) {
        count = count + 1;
        n = n / 10;
    }
    return count;
}
```

las pruebas

```c
assert(count_digits(7) == 1);
assert(count_digits(2026) == 4);
assert(count_digits(0) == 1);
assert(count_digits(-42) == 2);
```

Decidimos: el 0 tiene una cifra, y el signo no cuenta.

Corrección: tratar el 0 aparte, y seguir `while (n != 0)`.

terminal; Aborted lo escribe el shell, y puede variar

```
$ build/count_digits_bug
count_digits_bug: count_digits_bug.c:17: main: Assertion ‘count_digits(0) == 1’ failed.
Aborted
```

Qué casos elegir

Caso	Qué busca	En `count_digits`
Normal	Que la idea funcione	2026
Borde, a cada lado de un cambio	< por <=, salirse por uno	9 y 10
Menor valor, o cero	Ciclos que no llegan a correr	0
Negativos, si se aceptan	Suposiciones que no se dijeron	-42
Cada camino	Código que nunca se ejecutó	0 y otro

Dos cuidados

Pasar las pruebas no demuestra que no haya errores, solo que no los hay en esos casos.
El valor esperado se calcula a mano. No se copia de lo que dio el programa.

`assert` se puede apagar

`-D` define un nombre. Con `NDEBUG` definido, los `assert` desaparecen.

ndebug_demo.c

```c
assert(1 + 1 == 3);
printf("Llegué al final\n");
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -DNDEBUG ndebug_demo.c -o build/ndebug_off
$ build/ndebug_off
Llegué al final
```

Dos reglas. No validen la entrada con `assert`: para eso está el `if`. Y no pongan dentro nada que deba ocurrir siempre: con `assert(scanf(...) == 1)` apagado, tampoco se lee.

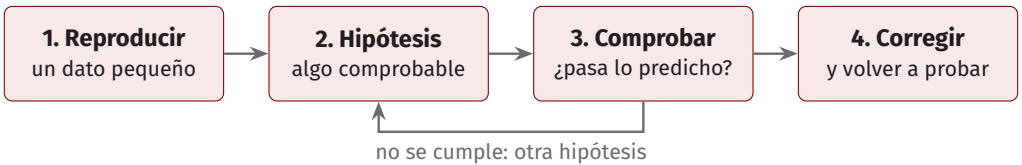
SECCIÓN 06

Depurar

06



Depurar con método



Herramienta	Actúa	Encuentra
`-Wall -Wextra`	Al compilar	Código sospechoso o inválido
`-fsanitize`	Mientras corre	Memoria y comportamiento indefinido, si ocurren
`assert`	Mientras corre	Resultados distintos de lo esperado
`gdb`	Cuando ustedes piden	Nada solo: deja mirar qué hace el programa

El caso: la suma de las cifras

digit_sum_bug.c

```c
int digit_sum(int n) {
    int sum = 0;
    while (n > 9) {
        int last = n % 10;
        sum = sum + last;
        n = n / 10;
    }
    return sum;
}
```

terminal

```
$ echo 407 | build/digit_sum_bug
La suma de las cifras de 407 es 7
$ echo 5 | build/digit_sum_bug
La suma de las cifras de 5 es 0
```

El caso: la suma de las cifras

digit_sum_bug.c

```c
int digit_sum(int n) {
    int sum = 0;
    while (n > 9) {
        int last = n % 10;
        sum = sum + last;
        n = n / 10;
    }
    return sum;
}
```

terminal

```
$ echo 407 | build/digit_sum_bug
La suma de las cifras de 407 es 7
$ echo 5 | build/digit_sum_bug
La suma de las cifras de 5 es 0
```

Hipótesis

Falta la cifra de la izquierda. **Predicción:** con 407, el ciclo termina con `n` en 4.

gdb: compilar y sus comandos

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -O0 digit_sum_bug.c -o build/digit_sum_bug_gdb
```

`-g`: guarda en el ejecutable los archivos, las líneas y los nombres de variables. `-O0`: sin optimizar, cada línea corre en orden. Sin `-fsanitize`: agrega código que aquí estorba.

`break digit_sum`	Punto de parada *(breakpoint)* en esa función, o en una línea: `break 11`
`run < entrada.txt`	Ejecuta hasta el siguiente punto de parada, leyendo del archivo
`next`	Ejecuta la línea completa, sin entrar en las llamadas
`step`	Ejecuta la línea, y entra en una llamada a una función nuestra
`print n`	Muestra el valor actual de `n`
`backtrace`	Muestra la pila de llamadas
`finish`	Sigue hasta que termine la función, y dice qué devolvió
`continue`, `quit`	Sigue hasta el próximo punto de parada o el final; sale

La primera vuelta, línea por línea

gdb

```
(gdb) break digit_sum
Breakpoint 1 at 0x830: file digit_sum_bug.c, line 5.
(gdb) run < entrada.txt
Breakpoint 1, digit_sum (n=407) at digit_sum_bug.c:5
5       int sum = 0;
(gdb) next
6       while (n > 9) {
(gdb) next
7           int last = n  % 10;
(gdb) next
8           sum = sum + last;
(gdb) print last
$2 = 7
(gdb) next
9           n = n / 10;
(gdb) print sum
$3 = 7
```

Muy importante

La línea que muestra **todavía no se ejecutó**: es la que viene.

Sesión real, recortada. Las direcciones como `0x830` dependen del entorno. Quitamos de aquí un `print n`, por eso `$2`.

Comprobar la predicción

gdb

```
(gdb) break 11
Breakpoint 2 at 0xaaaaaaaaa089c: file digit_sum_bug.c, line 11.
(gdb) continue
Continuing.

Breakpoint 2, digit_sum (n=4) at digit_sum_bug.c:11
11      return sum;
(gdb) print n
$4 = 4
(gdb) print sum
$5 = 7
(gdb) continue
Continuing.
La suma de las cifras de 407 es 7
```

La hipótesis se cumple: el ciclo terminó con `n` en 4 y `sum` en 7.

Corregir: `while (n > 0)`.

terminal

```
$ echo 407 | build/digit_sum
La suma de las cifras de 407 es 11
```

`step` y `backtrace`: la pila de verdad

gdb

```
(gdb) break main
(gdb) run
Breakpoint 1, main () at call_chain.c:15
15      int total = sum_of_squares(3, 4);
(gdb) step
sum_of_squares (a=3, b=4) at call_chain.c:9
9       int first = square(a);
(gdb) step
square (x=3) at call_chain.c:4
4       int result = x * x;
(gdb) backtrace
#0  square (x=3) at call_chain.c:4
#1  0x0000aaaaaaaa07e0 in sum_of_squares (a=3, b=4) at call_chain.c:9
#2  0x0000aaaaaaaa0818 in main () at call_chain.c:15
```

`#0` es el marco más reciente: `square` fue llamada por `sum_of_squares`, en su línea 9, y esta por `main`, en su línea 15. Sesión recortada; las direcciones `0x...` dependen del entorno.

SECCIÓN 07

Laboratorio

07

El laboratorio de hoy

1. `is_prime` e `is_leap_year` con `assert`, en `lab/functions.c`. No leen ni escriben. Casos para `is_prime`: 0, 1, 2, -7, 13, y los cuadrados perfectos 4 y 49.
2. `lab/grades_bug.c` compila sin advertencias y tiene **cuatro bugs**. Encuéntrenlos con `gdb`: compilen con `-g -O0` y corran con `run < lab/grades.txt`.
3. En ese programa: `break is_passing` y `backtrace`. Dibujen la pila. ¿Cuántos marcos hay?
4. Reorganicen `lab/adder4.c` con `sum_bit` y `carry_bit`, sin cambiar lo que escribe.

Antes de cada comando de `gdb`

Predigan qué debería salir. El bug está donde la predicción y la realidad se separan. Comprueben con `make test-lab`.

Para la próxima clase

- En el laboratorio: **Práctica 2**. Las cuatro partes son trabajo autónomo.
- La próxima semana es el **examen parcial**, de la semana 1 a esta.

Preguntas

¿Qué escribe `inc(a); printf("%d", a);` con `a` en 1, si `inc` hace `x = x + 1`?
¿Qué casos de prueba elegirían para `is_prime`, y por qué esos?
Dibujen la pila cuando `main` llama a `f` y `f` llama a `g`.