

Introducción a la Programación

Lo esencial de C, otra vez

Tipos, control, funciones, `assert` y advertencias

Gerson Garrido

Semana 1

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. De qué trata este curso
2. Repaso rápido: tipos, operadores, ciclos y funciones
3. Separar el cálculo de la entrada y salida
4. Probar con `assert`: elegir los casos borde
5. Leer las advertencias como si fueran errores
6. El laboratorio de esta semana

> **Lectura**
>
> Apuntes de la semana 1. Código en `code/week-01`.

SECCIÓN 01

De qué trata este curso

01

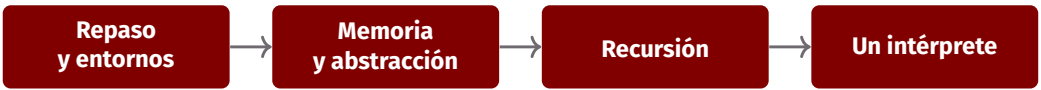
Un programa grande pide otras herramientas

Problema	Lo que aprenderán
No se puede entender todo a la vez	Leer con método, interfaces, módulos
No se sabe qué pasa en la memoria	Modelo de entornos, punteros, memoria dinámica
Un cambio aquí rompe algo allá	Pruebas que se corren después de cada cambio
El error está lejos de donde se nota	Depurar con método: `gdb` y sanitizers
El problema es muy difícil de frente	Descomponer: recursión, backtracking

Idea clave

Casi siempre se lee más código del que se escribe. Antes de cambiar un programa, hay que entenderlo.

Cómo está armado



- 2 h de teoría y 4 h de laboratorio a la semana, en dos sesiones.
- **El laboratorio es el centro.** Traen código que ustedes leen y extienden, con sus pruebas.
- No hay libro: los apuntes son el texto. Calendario y políticas: en el programa del curso.

SECCIÓN 02

Tipos y operadores

02



Tres tipos, tres recordatorios

terminal

```
$ build/types
int: 4 bytes, de -2147483648 a 2147483647
double: 8 bytes
char: 1 byte
7 tiene el código 55 y vale 7
0.30000000000000004
x no es igual a 0.3
```

- **`char` es un número pequeño:** `digit - '0'` da el valor de la cifra.
- **`double` es aproximado:** nunca se comparan con `==`. Para dinero, céntimos en un `int`.
- **`int` tiene un límite:** y `-INT_MIN` no cabe en un `int`.

Overflow: comportamiento indefinido

overflow.c

```c
int big = INT_MAX;
big = big + 1;
```

terminal

```
$ build/overflow
overflow.c:7:9: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'
```

El lenguaje no promete nada

El programa sigue, pero C no promete qué valor queda. El sanitizer lo avisa mientras corre, solo si esa línea se ejecuta.

¿Qué escribe?

division.c

```c
int a = 7, b = 2;
printf("%d %.1f\n", a / b, (double) a / b);
```

terminal

```
$ build/division
3 3.5
0.0 0.5
3
```

- Entre dos `int`, `/` es división entera. Da 3.
- El cast va primero que `/`: la división es entre `double`.

La trampa de siempre

division.c

```c
double half = 1 / 2;
double real_half = 1.0 / 2;
int truncated = 3.99;
```

- `half` es `double`, pero `1 / 2` se calcula antes, entre enteros: 0.
- Basta un operando `double`: 0.5.
- De `double` a `int` se descartan los decimales: 3, no 4.

> **Idea clave**
>
> Lo que manda es el tipo de los **operandos**, no el de la variable que recibe.

Con negativos, cuidado

operators.c

```c
printf("%d %d\n", 7 / 2, -7 / 2);
printf("%d %d\n", 7 % 3, -7 % 3);
printf("%d %d\n", 2 + 3 * 4, (2 + 3) * 4);
```

terminal

```
$ build/operators
3 -3
1 -1
14 20
```

- La división entera se acerca a 0: `-7 / 2` da -3.
- El resto lleva el signo del dividendo: `-7 % 3` da -1.
- Si `n` es negativo, `n % 10` es negativo o 0. Una cuenta que funciona con positivos puede fallar en silencio.

Cortocircuito

operators.c

```c
int count = 0;
int total = 50;

if (count != 0 && total / count > 10) {
```

- `&&` no calcula su lado derecho si el izquierdo ya es falso.
- `||` no lo calcula si el izquierdo ya es verdadero.
- Por eso no se divide entre 0. Con los lados al revés, sí.

SECCIÓN 03

Ciclos

03

Cada ciclo le dice algo al lector

Ciclo	Cuándo	Ejemplo
`for`	Se sabe cuántas vueltas da	Recorrer de 1 a `n`
`while`	Depende de una condición	Dividir hasta llegar a 1
`do while`	Corre al menos una vez	Contar cifras: el 0 tiene una

Tres preguntas para cada ciclo

¿Con qué valor empieza? ¿Con qué valor termina? ¿Qué pasa si no da ninguna vuelta?

`do while`: el 0 tiene una cifra

loops.c

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

terminal

```
$ build/loops
3
6
4 1 2
```

La última línea: `2026`, `0` y `-42`.

`n != 0` sirve con negativos, porque la división se acerca a 0. Con `n > 0`, el -42 daría una sola vuelta.

Salirse por uno

off_by_one.c

```c
for (int i = 1; i < n; i++) {
```

terminal

```
$ build/off_by_one
2
3
```

Con 9 da 2, y deberían ser 3. Con 10 da 3, y es correcto. **El mismo bug acierta con un dato y falla con otro.**

El ciclo nunca mira al propio `n`



`scanf` devuelve cuántos datos leyó

grades_average.c

```c
while (scanf("%d", &grade) == 1) {
    total = total + grade;
    count = count + 1;
}
```

terminal

```
$ echo 12 8 15 | build/grades_average
3 notas, promedio 11.67
$ build/grades_average < /dev/null
No hay notas
```

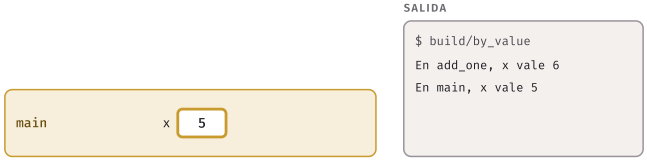
Termina al acabarse la entrada o al llegar algo que no es un número. A mano: `Ctrl+D`.

SECCIÓN 04

Funciones

04

Un parámetro siempre recibe una copia



add_one retornó: su marco desapareció, y la x de main sigue en 5

terminal

```
$ build/by_value
En add_one, x vale 6
En main, x vale 5
En try_swap, a = 2 y b = 1
En main, first = 1 y second = 2
```

Lo que se olvida de las funciones

by_value.c

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

- `return` entrega **un solo** valor. Para dos resultados: punteros, semana 4.
- `return` termina de inmediato.
- Si no es `void`, debe devolver algo en **todos** los caminos.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/double_argument.c -o build/double_argument
$ build/double_argument
4
```

Con `square(2.5)`: C convierte el argumento sin avisar. Recibe un 2.

SECCIÓN 05

Separar el cálculo

05

El ejemplo: dar un vuelto

pseudocódigo

```
algoritmo monedas_necesarias(centimos)
    si centimos < 0 o centimos mod 10 ≠ 0 entonces
        devolver -1
    fin si
    monedas ← 0
    mientras centimos > 0 hacer
        centimos ← centimos - moneda_mayor(centimos)
        monedas ← monedas + 1
    fin mientras
    devolver monedas
fin algoritmo
```

Monedas de 10, 20 y 50 céntimos y de 1, 2 y 5 soles. Contamos en céntimos.

Antes	Moneda	Queda	coins
380	200	180	1
180	100	80	2
80	50	30	3
30	20	10	4
10	10	0	5

Primera versión: todo junto

change_print.c

```c
int left = cents;
int coins = 0;
while (left > 0) {
    if (left >= 500) {
        left = left - 500;
    } else if (left >= 200) {
...
```

terminal

```
$ echo 380 | build/change_print
Para 380 céntimos hacen falta 5 monedas
```

Funciona. Pero, ¿cómo sabemos que da bien con 10, 500, 490 y -10?

- Hay que correrlo una vez por cada dato y leer la frase.
- Nadie lo repite después de cada cambio.
- El resultado se va a la pantalla: ningún código lo puede comparar.

Segunda versión: solo calcula

change.c

```c
int largest_coin(int cents) {
    if (cents >= 500) {
        return 500;
    } else if (cents >= 200) {
        return 200;
    } else if (cents >= 100) {
        return 100;
    } else if (cents >= 50) {
        return 50;
    } else if (cents >= 20) {
        return 20;
    }
    return 10;
}
```

change.c

```c
int coins_needed(int cents) {
    if (cents < 0 || cents % 10 != 0) {
        return -1;
    }

    int coins = 0;
    while (cents > 0) {
        cents = cents - largest_coin(
      cents);
        coins = coins + 1;
    }
    return coins;
}
```

Ninguna lee ni escribe. Cambiar `cents` no molesta: es una copia.

`main` queda con la entrada y la salida

change.c

```c
int coins = coins_needed(cents);
if (coins < 0) {
    printf("No se puede dar %d céntimos en monedas\n
      ", cents);
    return 1;
}

printf("Para %d céntimos hacen falta %d monedas\n",
       cents, coins);
```

terminal

```
$ echo 380 | build/change
Para 380 céntimos hacen falta 5 monedas
$ echo 15 | build/change
No se puede dar 15 céntimos en monedas
```

Idea clave

Lectura y escritura en `main`. Cálculos en funciones.

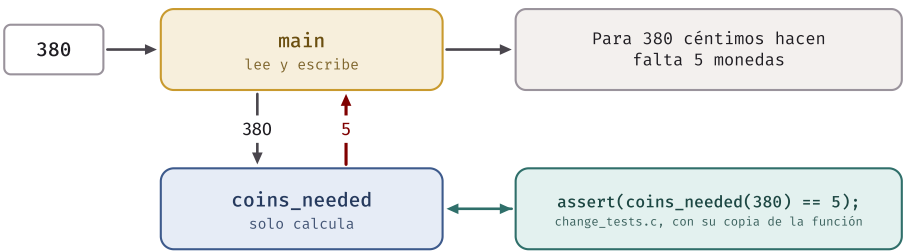
Lo que se gana al separar

CHANGE_PRINT.C: TODO JUNTO



el resultado solo es una frase en la pantalla: ningún código lo puede comparar

CHANGE.C: EL CÁLCULO APARTE



SECCIÓN 06

Probar con `assert`

06

Una prueba: un dato cuyo resultado conocemos

change_tests.c

```c
assert(coins_needed(380) == 5);     // caso normal: 200, 100, 50, 20, 10
assert(coins_needed(0) == 0);       // el menor monto: no se da ninguna moneda
assert(coins_needed(10) == 1);      // la moneda más chica
assert(coins_needed(500) == 1);     // justo una moneda grande
assert(coins_needed(490) == 5);     // justo debajo de 500
assert(coins_needed(1000) == 2);    // más de una moneda grande
assert(coins_needed(15) == -1);     // no es múltiplo de 10
assert(coins_needed(-10) == -1);    // negativo
```

terminal

```
$ build/change_tests
Todas las pruebas pasaron
```

Un solo carácter de diferencia

change_bug.c

```
1  if (cents > 500) {
```

terminal

```
$ build/change_bug
change_bug: change_bug.c:37: main: Assertion ‘coins_needed(500) == 1’ failed.
```

- Pasaron 380, 0 y 10. Falló la primera prueba que miraba el borde.
- Con 500, el bug da 200, 200 y 100: tres monedas. Un resultado creíble.
- El mensaje da el archivo, la línea, la función y la condición.

Elegir los casos

Tipo de caso	Por ejemplo	Qué busca
Normal	Un dato común	Que la idea sea correcta
Cero	0	Un ciclo que no corre, dividir entre 0
Límite de una condición	499, 500, 501	`<` en lugar de `<=`
El menor o el mayor	`1`, `INT_MAX`, `INT_MIN`	Overflow
Negativos	-1, -7	Resto y división con signo
Datos que no valen	15 céntimos	Que responda lo acordado
Un caso por camino	Cada rama del `if`	Una rama que nunca se probó

Ojo

Las pruebas no demuestran que el programa esté bien: muestran que no falló con esos datos.

¿Qué casos elegirían para `is_prime`?

`n`	¿Primo?	Qué error descubre
13	sí	La idea general
2	sí	Un ciclo que empieza mal y descarta al menor primo
1, 0	no	Un ciclo que no corre y deja la respuesta en “primo”
-7	no	Olvidar los negativos
4, 49	no	Parar antes de la raíz, en un cuadrado perfecto
`INT_MAX`	sí	`i * i <= n`: el producto no cabe. Mejor `i <= n / i`

`assert` no es para todo

Error común

No validen la entrada del usuario con `assert`. Que alguien escriba `abc` no es un error del programador: se responde con un mensaje.

- `assert` se apaga al compilar con `-DNDEBUG`, y esas líneas desaparecen enteras.
- Por eso, nunca `assert(scanf("%d", &n) == 1)`: sin `assert`, tampoco hay lectura.

SECCIÓN 07

Leer las advertencias

07

Una función sin `return`

errors/missing_return.c

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

- Una advertencia no detiene al compilador.
- Debía dar 1 o 0. Salió 5 en esta corrida; en otra computadora, quizá otra cosa.
- **Corrió y dio un número creíble.** El sanitizer no vigila este caso.

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]
    8 | }
      | ^
$ build/missing_return
5
```

Asignar en vez de comparar

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/assign_in_if.c -o build/assign_in_if
errors/assign_in_if.c:7:9: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
    7 |     if (grade = 20) {
      |         ^~~~~
$ build/assign_in_if
Nota perfecta: 20
```

La nota era 12. Tomó el camino equivocado, y además cambió la variable.

Comparaciones encadenadas

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/chained_compare.c -o build/chained_compare
errors/chained_compare.c:7:11: warning: comparisons like ‘X<=Y<=Z’ do not have their mathematical meaning [-Wparentheses]
    7 |     if (0 <= grade <= 20) {
      |         ~~^~~~~~~
$ build/chained_compare
Nota válida
```

La nota era 25. Se calcula `0 <= grade`, que da 0 o 1, y eso se compara con 20. Se escribe `0 <= grade && grade <= 20`.

Formato equivocado

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/wrong_format.c -o build/wrong_format
errors/wrong_format.c:7:24: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘double’ [-Wformat=]
    7 |     printf("Promedio: %d\n", average);
      |                       ~^     ~~~~~~~
      |                        |     |
      |                        int   double
      |                       %f
```

Dice qué esperaba, qué recibió, y hasta sugiere la corrección.

Por qué también `-Wextra`

terminal

```
$ gcc -Wall -std=c17 errors/unused_parameter.c -o build/unused_parameter
$ gcc -Wall -Wextra -std=c17 errors/unused_parameter.c -o build/unused_parameter
errors/unused_parameter.c:4:31: warning: unused parameter ‘step’ [-Wunused-parameter]
    4 | int next_value(int value, int step) {
      |                           ~~~~^~~~
```

`-Wall` solo no avisa. A pesar del nombre, no activa todas las advertencias.

Las advertencias no lo ven todo

errors/uninitialized.c

```c
int total;
...
for (int i = 1; i <= 3; i++) {
    total = total + i;
}
```

Debía dar 6. En esta corrida dio 8, y otra computadora puede dar otra cosa. Sin `-O2`, `gcc` no avisó.

Que no haya advertencias no significa que no haya errores.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/uninitialized.c -o build/uninitialized
$ build/uninitialized
8
$ gcc -Wall -Wextra -std=c17 -O2 errors/uninitialized.c -o build/uninitialized_o2
errors/uninitialized.c:8:15: warning: ‘total’ is used uninitialized [-Wuninitialized]
```

Cada herramienta ve algo distinto

Herramienta	Cuándo actúa	Qué encuentra
`-Wall -Wextra`	Al compilar	Código sospechoso, sin correr nada
Sanitizer	Mientras corre	Overflow, accesos fuera de un arreglo…
`assert`	Mientras corre	Resultados distintos de los esperados
`gdb`	Cuando lo usan	Nada solo: deja mirar el programa por dentro

Regla del curso

Cada advertencia se lee y se corrige, empezando por la primera. Un error temprano suele provocar otros después.

SECCIÓN 08

El laboratorio

08

Entorno, `make` y Git

`env/check.sh`	Revisa el entorno
`make`	Compila en `build/`
`make test`	Comprueba los ejemplos
`make test-lab`	Prueba su laboratorio
`make clean`	Borra lo generado

`git clone` copia un repositorio con todo su historial.

terminal

```
$ git clone proyecto copia
Cloning into 'copia'...
done.
$ cd copia
$ git commit -m "Agrega una nota"
[main a5079b5] Agrega una nota
 1 file changed, 1 insertion(+)
$ git log --oneline
a5079b5 Agrega una nota
f81b155 Primera nota
$ cd ../proyecto
$ git log --oneline
f81b155 Primera nota
```

Los identificadores serán otros. El commit nuevo solo está en `copia`.

Cada copia tiene su propio historial



Sesión 1: entorno y prueba diagnóstica

1. `env/check.sh`, `make`, `make test`, y la sesión de Git de los apuntes.
2. Prueba diagnóstica en `lab/diagnostic.c`. Se comprueba con `make test-diagnostic`.

Función	Revisa	Si falla, repasen
`average_of_two`	División y conversión	Semana 2
`digit_value`	`char`, ASCII	Semanas 2 y 3
`clamp`	`if`, `else if`, bordes	Semana 4
`sum_to`	Ciclos, salirse por uno	Semana 5
`dice_ways`	Ciclos anidados	Semana 6

No tiene nota

Sirve para que cada uno sepa qué repasar. Háganla con honestidad.

Sesión 2: funciones con sus pruebas

- `digit_sum`: el signo no cuenta, y debe servir con `INT_MIN`.
- `is_prime`: con los casos de hoy, también `INT_MAX`.
- `gcd`: nunca negativo. Con `(12, -18)`, el algoritmo tal cual termina en -6.
- `collatz_steps`: con 6 da 8 pasos. Se cuentan las flechas, no los números.



La pregunta final

¿Cuál es el número menor que 10 000 con más pasos? Escríbanlo con `longest_collatz` y corran `build/lab/collatz`.

Para la próxima clase

- Terminar las dos sesiones: `make test-lab` debe pasar todo.
- Repasar lo que señaló la prueba diagnóstica.
- La próxima semana: arreglos, cadenas, `struct`, archivos y cómo leer código ajeno.

Preguntas

¿Qué escribe `printf("%d %.1f", a / b, (double) a / b)` con `a` en 7 y `b` en 2? ¿Por qué una función que calcula y además imprime es más difícil de probar? ¿Qué advertencia da `-Wall` si una función que devuelve `int` no tiene `return`, y por qué importa?