

Introducción a la Computación

Variables y programas secuenciales

Guardar, calcular, leer y escribir

Gerson Garrido

Semana 2

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Variables: un nombre para un lugar en memoria
2. Tipos: `int`, `double`, `char` y `sizeof`
3. Operadores, precedencia y división entera
4. Conversiones y cast
5. Escribir con `printf` y leer con `scanf`
6. De la fórmula al programa, `math.h` y archivos

> **Lectura**
>
> Apuntes de la semana 2. El código está en `code/week-02`.

Programas secuenciales

- La semana pasada: un programa que siempre escribe lo mismo.

Programas secuenciales

- La semana pasada: un programa que siempre escribe lo mismo.
- Esta semana: programas que **guardan**, **calculan** y **leen**.

Programas secuenciales

- La semana pasada: un programa que siempre escribe lo mismo.
- Esta semana: programas que **guardan**, **calculan** y **leen**.
- **Secuencial**: las instrucciones se ejecutan una tras otra, de arriba hacia abajo, cada una una vez.

Programas secuenciales

- La semana pasada: un programa que siempre escribe lo mismo.
- Esta semana: programas que **guardan**, **calculan** y **leen**.
- **Secuencial**: las instrucciones se ejecutan una tras otra, de arriba hacia abajo, cada una una vez.
- Todavía no hay decisiones (semana 4) ni ciclos (semana 5).

Programas secuenciales

- La semana pasada: un programa que siempre escribe lo mismo.
- Esta semana: programas que **guardan**, **calculan** y **leen**.
- **Secuencial**: las instrucciones se ejecutan una tras otra, de arriba hacia abajo, cada una una vez.
- Todavía no hay decisiones (semana 4) ni ciclos (semana 5).

Meta de hoy

El vuelto de S/ 7.80 que siguieron a mano, ahora para cualquier monto.

SECCIÓN 01

Variables

01



Un nombre para un lugar en memoria

Variable *(variable)*

Un nombre para un lugar en memoria donde se guarda un valor.

- Se **declara** con un tipo y un nombre.
- Normalmente recibe un valor inicial.

```
int balance = 10;
```



Un programa con una variable

balance.c

```c
#include <stdio.h>

int main(void) {
    int balance = 10;
    printf("Saldo inicial: %d\n", balance);

    balance = balance - 3;
    printf("Después del pasaje: %d\n", balance);

    balance = balance + 5;
    printf("Después de la recarga: %d\n", balance);

    return 0;
}
```

Línea	balance

Un programa con una variable

balance.c

```c
#include <stdio.h>

int main(void) {
    int balance = 10;
    printf("Saldo inicial: %d\n", balance);

    balance = balance - 3;
    printf("Después del pasaje: %d\n", balance);

    balance = balance + 5;
    printf("Después de la recarga: %d\n", balance);

    return 0;
}
```

Línea	balance
declaración	10

Un programa con una variable

balance.c

```c
#include <stdio.h>

int main(void) {
    int balance = 10;
    printf("Saldo inicial: %d\n", balance);

    balance = balance - 3;
    printf("Después del pasaje: %d\n", balance);

    balance = balance + 5;
    printf("Después de la recarga: %d\n", balance);

    return 0;
}
```

Línea	balance
declaración	10
- 3	7

Un programa con una variable

balance.c

```c
#include <stdio.h>

int main(void) {
    int balance = 10;
    printf("Saldo inicial: %d\n", balance);

    balance = balance - 3;
    printf("Después del pasaje: %d\n", balance);

    balance = balance + 5;
    printf("Después de la recarga: %d\n", balance);

    return 0;
}
```

Línea	balance
declaración	10
- 3	7
+ 5	12

Un programa con una variable

balance.c

```c
#include <stdio.h>

int main(void) {
    int balance = 10;
    printf("Saldo inicial: %d\n", balance);

    balance = balance - 3;
    printf("Después del pasaje: %d\n", balance);

    balance = balance + 5;
    printf("Después de la recarga: %d\n", balance);

    return 0;
}
```

Línea	`balance`
declaración	10
- 3	7
+ 5	12

`%d` marca un hueco: ahí se escribe el valor del `int`.

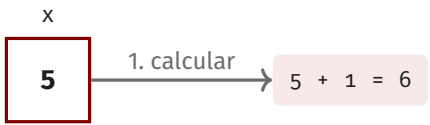
Asignar no es igualar

```
X = X + 1;
```



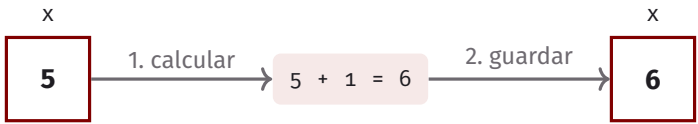
Asignar no es igualar

```
x = x + 1;
```



Asignar no es igualar

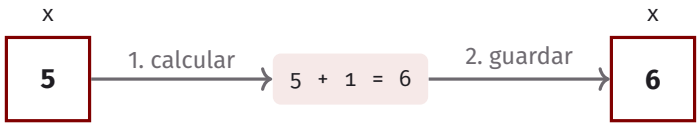
```
x = x + 1;
```



- Primero se calcula la **derecha**, con los valores actuales.
- Después se guarda en la **izquierda**. Lo anterior se pierde.

Asignar no es igualar

```
x = x + 1;
```



- Primero se calcula la **derecha**, con los valores actuales.
- Después se guarda en la **izquierda**. Lo anterior se pierde.
- En C, `=` asigna. Para comparar se usa `==` (semana 4).
- En pseudocódigo: $x \leftarrow x + 1$.

Una asignación no deja una fórmula

not_a_formula.c

```c
int price = 2;
int total = price * 3;

price = 10;
```

terminal

```
$ build/not_a_formula
price = 10
total = 6
```

Una película, no una foto

Cada instrucción usa los valores que las variables tienen en ese momento. `total` recibió un 6, no la fórmula. Una hoja de cálculo se actualiza sola; C, no.

El orden importa: ¿intercambiar dos valores?

Instrucción	`left`	`right`
(al inicio)	4	9

El orden importa: ¿intercambiar dos valores?

Instrucción	`left`	`right`
(al inicio)	4	9
`left = right;`	9	9

El orden importa: ¿intercambiar dos valores?

Instrucción	`left`	`right`
(al inicio)	4	9
`left = right;`	9	9
`right = left;`	9	9

El orden importa: ¿intercambiar dos valores?

Instrucción	`left`	`right`
(al inicio)	4	9
`left = right;`	9	9
`right = left;`	9	9

No funciona

La primera asignación destruyó el 4. Cuando la segunda lo necesita, ya no existe.

Intercambiar con una tercera variable

pseudocódigo

```
algoritmo intercambiar()
    left ← 4
    right ← 9
    saved ← left
    left ← right
    right ← saved
    escribir left, right
fin algoritmo
```

swap_values.c

```c
1  int saved = left;
2  left = right;
3  right = saved;
```

	left	right	saved
inicio	4	9	

Intercambiar con una tercera variable

pseudocódigo

```
algoritmo intercambiar()
    left ← 4
    right ← 9
    saved ← left
    left ← right
    right ← saved
    escribir left, right
fin algoritmo
```

swap_values.c

```c
1  int saved = left;
2  left = right;
3  right = saved;
```

	left	right	saved
inicio	4	9	
1.	4	9	4

Intercambiar con una tercera variable

pseudocódigo

```
algoritmo intercambiar()
    left ← 4
    right ← 9
    saved ← left
    left ← right
    right ← saved
    escribir left, right
fin algoritmo
```

swap_values.c

```c
1  int saved = left;
2  left = right;
3  right = saved;
```

	left	right	saved
inicio	4	9	
1.	4	9	4
2.	9	9	4

Intercambiar con una tercera variable

pseudocódigo

```
algoritmo intercambiar()
    left ← 4
    right ← 9
    saved ← left
    left ← right
    right ← saved
    escribir left, right
fin algoritmo
```

swap_values.c

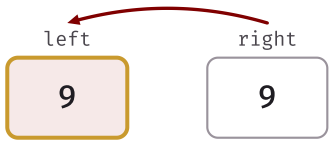
```
1  int saved = left;
2  left = right;
3  right = saved;
```

	left	right	saved
inicio	4	9	
1.	4	9	4
2.	9	9	4
3.	9	4	4

Cada asignación copia y reemplaza

SIN TERCERA VARIABLE

```
left = right;
right = left;
```



el 4 ya no está en ninguna parte

CON UNA TERCERA VARIABLE

```
int saved = left;
left = right;
right = saved;
```

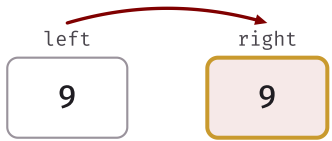


left = right copia el 9 en left, y lo que había se pierde

Cada asignación copia y reemplaza

SIN TERCERA VARIABLE

```
left = right;
right = left;
```



el 4 ya no está en ninguna parte

CON UNA TERCERA VARIABLE

```
int saved = left;
left = right;
right = saved;
```



right = left copia otro 9: las dos variables quedan en 9

Cada asignación copia y reemplaza

SIN TERCERA VARIABLE

```
left = right;
right = left;
```



el 4 ya no está en ninguna parte

CON UNA TERCERA VARIABLE

```
int saved = left;
left = right;
right = saved;
```



saved recibe una copia del 4 antes de que se pierda

Cada asignación copia y reemplaza

SIN TERCERA VARIABLE

```
left = right;
right = left;
```



el 4 ya no está en ninguna parte

CON UNA TERCERA VARIABLE

```
int saved = left;
left = right;
right = saved;
```



ahora left ya puede recibir el 9

Cada asignación copia y reemplaza

SIN TERCERA VARIABLE

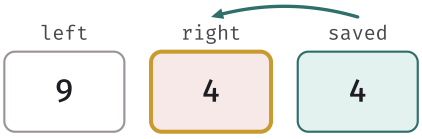
```
left = right;
right = left;
```



el 4 ya no está en ninguna parte

CON UNA TERCERA VARIABLE

```
int saved = left;
left = right;
right = saved;
```



los dos valores cambiaron de lugar

right recibe el 4 que estaba guardado en saved

Error común: usar una variable sin darle valor

terminal

```
$ gcc -Wall -Wextra -std=c17 broken/uninitialized.c -o build/uninitialized
broken/uninitialized.c: In function ‘main’:
broken/uninitialized.c:7:5: warning: ‘total’ is used uninitialized [-Wuninitialized]
    7 |     printf("total = %d\n", total);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
broken/uninitialized.c:5:9: note: ‘total’ was declared here
    5 |     int total;
      |         ^~~~~
```

Comportamiento indefinido ***(undefined behavior)***

No vale 0: el lenguaje no dice qué debe pasar. A veces parece funcionar. Sin `-Wall`, `gcc` no dice nada, y `-fsanitize` tampoco lo detecta.

Regla: den un valor a cada variable al declararla, y arreglen todos los avisos.

SECCIÓN 02

Tipos de datos

02

Tres tipos

Tipo	Guarda	Ejemplos	Para qué
`int`	enteros	`20`, `0`, `-3`	cantidades, años
`double`	números con decimales	`3.5`, `2.0`	medidas, promedios
`char`	un carácter	`'A'`, `'7'`	una letra, un símbolo

Tres tipos

Tipo	Guarda	Ejemplos	Para qué
`int`	enteros	`20`, `0`, `-3`	cantidades, años
`double`	números con decimales	`3.5`, `2.0`	medidas, promedios
`char`	un carácter	`'A'`, `'7'`	una letra, un símbolo

- `7` es `int`. `7.0` es `double`. `'7'` es `char`. `"7"` es una cadena *(string)*.

Tres tipos

Tipo	Guarda	Ejemplos	Para qué
`int`	enteros	`20`, `0`, `-3`	cantidades, años
`double`	números con decimales	`3.5`, `2.0`	medidas, promedios
`char`	un carácter	`'A'`, `'7'`	una letra, un símbolo

- `7` es `int`. `7.0` es `double`. `'7'` es `char`. `"7"` es una cadena *(string)*.
- El tipo dice cuántos bytes usa la variable y cómo interpretarlos.

Tres tipos

Tipo	Guarda	Ejemplos	Para qué
`int`	enteros	`20`, `0`, `-3`	cantidades, años
`double`	números con decimales	`3.5`, `2.0`	medidas, promedios
`char`	un carácter	`'A'`, `'7'`	una letra, un símbolo

- `7` es `int`. `7.0` es `double`. `'7'` es `char`. `"7"` es una cadena *(string)*.
- El tipo dice cuántos bytes usa la variable y cómo interpretarlos.
- También existe `float`, con menos precisión. Usamos `double`.

`sizeof`: cuántos bytes ocupan

sizes.c

```c
printf("int: %zu bytes\n", sizeof(int))
    ;
```

terminal

```
$ build/sizes
int: 4 bytes
double: 8 bytes
char: 1 bytes
la variable age: 4 bytes
age vale 19
```

- `%zu` es el formato de `sizeof`.
- Solo `char` está fijado por el lenguaje. Los otros son los de este entorno.

`sizeof`: cuántos bytes ocupan

sizes.c

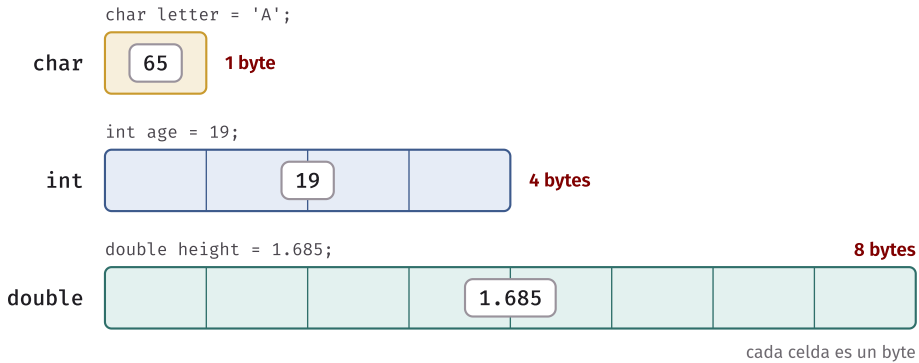
```
1  printf("int: %zu bytes\n", sizeof(int))
       ;
```

terminal

```
$ build/sizes
int: 4 bytes
double: 8 bytes
char: 1 bytes
la variable age: 4 bytes
age vale 19
```

- `%zu` es el formato de `sizeof`.
- Solo `char` está fijado por el lenguaje. Los otros son los de este entorno.
- Tamaño fijo significa límite:
  - `int` tiene un máximo. Lo que no cabe es un overflow, un error del programa.
  - `double` guarda una aproximación.
- Lo explica la semana 3.

Los tres tipos, a escala



El valor es de la variable completa: no se reparte una cifra por celda.

`char` es un número pequeño

char_codes.c

```c
char letter = 'A';
char digit = '7';

printf("%c tiene el código %d\n", letter, letter);
printf("La letra siguiente es %c\n", letter + 1);
printf("%c tiene el código %d\n", digit, digit);
printf("El valor numérico de %c es %d\n",
       digit, digit - '0');
```

terminal

```
$ build/char_codes
A tiene el código 65
La letra siguiente es B
7 tiene el código 55
El valor numérico de 7 es 7
```

`'7'` no es el número 7: su código es 55. La tabla de códigos se llama ASCII, y la vemos la próxima semana.

Una ñ no cabe en un `char`

terminal

```
$ gcc -Wall -Wextra -std=c17 broken/accent_char.c -o build/accent_char
broken/accent_char.c: In function ‘main’:
broken/accent_char.c:5:19: warning: multi-character character constant [-Wmultichar]
    5 |     char letter = ’ñ’;
      |                   ^~~
```

- La ñ y las vocales con tilde ocupan más de un byte en nuestros archivos.
- Mostramos el primer aviso. Hay un segundo, cuyo texto cambia según el procesador.
- En una cadena sí pueden ir: `printf("Sección")` funciona.
- La semana 3 explica cómo se guarda la ñ.

Constantes

igv_price.c

```c
const double IGV_RATE = 0.18;
double price = 0.0;

printf("Precio sin IGV: ");
scanf("%lf", &price);

double igv = price * IGV_RATE;
double total = price + igv;
```

terminal

```
$ build/igv_price
Precio sin IGV: 50
IGV: 9.00
Total: 59.00
```

- `const`: asignarle otro valor es un error de compilación.
- Nombre en mayúsculas.
- Un número con nombre se entiende mejor.

SECCIÓN 03

Operadores

03



Precedencia

Nivel	Operadores	Orden
1	`( )`	de adentro afuera
2	`-x`, `(double)`	
3	`* / %`	izquierda a derecha
4	`+ -`	izquierda a derecha
5	`=`	

- `%` es el **resto** de la división.
- Tiene el mismo nivel que `*` y `/`.

Precedencia

Nivel	Operadores	Orden
1	`( )`	de adentro afuera
2	`-x`, `(double)`	
3	`* / %`	izquierda a derecha
4	`+ -`	izquierda a derecha
5	`=`	

- `%` es el **resto** de la división.
- Tiene el mismo nivel que `*` y `/`.
- `20 / 2 * 5` da 50, no 2.

Precedencia

Nivel	Operadores	Orden
1	`( )`	de adentro afuera
2	`-x`, `(double)`	
3	`* / %`	izquierda a derecha
4	`+ -`	izquierda a derecha
5	`=`	

- `%` es el **resto** de la división.
- Tiene el mismo nivel que `*` y `/`.
- `20 / 2 * 5` da 50, no 2.
- Cuando tengan dudas, paréntesis. No cuestan nada.

Evaluar paso a paso

terminal

```
$ build/arithmetic
10 - 2 * 3 = 4
(10 - 2) * 3 = 24
20 / 2 * 5 = 50
10 - 4 - 3 = 3
7 + 8 % 3 * 2 = 11
```

```
7 + 8 % 3 * 2
```

Evaluar paso a paso

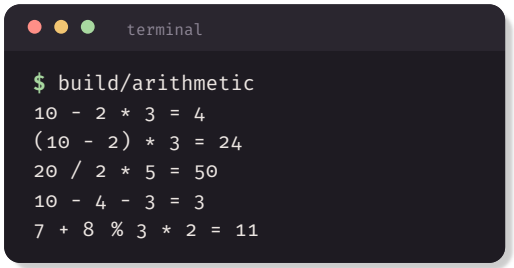
terminal

```
$ build/arithmetic
10 - 2 * 3 = 4
(10 - 2) * 3 = 24
20 / 2 * 5 = 50
10 - 4 - 3 = 3
7 + 8 % 3 * 2 = 11
```

```
7 + 8 % 3 * 2
7 + 2 * 2
```

`%` y `*` tienen igual nivel: va primero el de la izquierda.

Evaluar paso a paso




```
7 + 8 % 3 * 2
7 + 2 * 2
7 + 4
```

`%` y `*` tienen igual nivel: va primero el de la izquierda.

Evaluar paso a paso

terminal

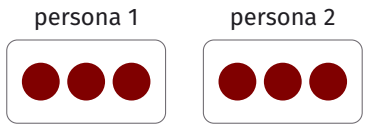
```
$ build/arithmetic
10 - 2 * 3 = 4
(10 - 2) * 3 = 24
20 / 2 * 5 = 50
10 - 4 - 3 = 3
7 + 8 % 3 * 2 = 11
```

```
7 + 8 % 3 * 2
7 + 2 * 2
7 + 4
11
```

`%` y `*` tienen igual nivel: va primero el de la izquierda.

División entera: `7 / 2` vale 3

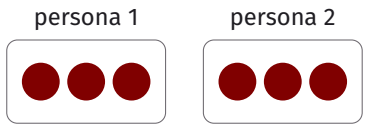
- Dos **enteros**: división entera. Los decimales se descartan, no se redondea.



7 caramelos entre 2 personas, sin partir ninguno.

División entera: `7 / 2` vale 3

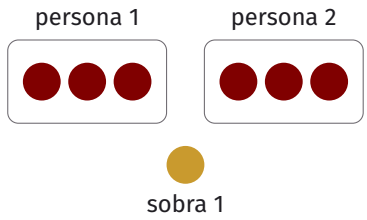
- Dos **enteros**: división entera. Los decimales se descartan, no se redondea.
- Al menos un **`double`**: la división de siempre. `7.0 / 2` vale 3.5.



7 caramelos entre 2 personas, sin partir ninguno.

División entera: `7 / 2` vale 3

- Dos **enteros**: división entera. Los decimales se descartan, no se redondea.
- Al menos un **`double`**: la división de siempre. `7.0 / 2` vale 3.5.
- El sobrante lo da `%`: `7 % 2` vale 1.



7 caramelos entre 2 personas, sin partir ninguno.

Con precisión: negativos y resto

terminal

```
$ build/division
17 / 5 = 3
17 % 5 = 2
17.0 / 5 = 3.400000
17 / 5.0 = 3.400000
-17 / 5 = -3
-17 % 5 = -2
17 % -5 = 2
```

- La división entera se acerca a **cero**: `-17 / 5` da -3, no -4.

Con precisión: negativos y resto

terminal

```
$ build/division
17 / 5 = 3
17 % 5 = 2
17.0 / 5 = 3.400000
17 / 5.0 = 3.400000
-17 / 5 = -3
-17 % 5 = -2
17 % -5 = 2
```

- La división entera se acerca a **cero**: `-17 / 5` da -3, no -4.
- Siempre se cumple: `(a / b) * b + a % b` vale `a`.
- Por eso el resto toma el signo del primer operando.

Con precisión: negativos y resto

terminal

```
$ build/division
17 / 5 = 3
17 % 5 = 2
17.0 / 5 = 3.400000
17 / 5.0 = 3.400000
-17 / 5 = -3
-17 % 5 = -2
17 % -5 = 2
```

- La división entera se acerca a **cero**: `-17 / 5` da -3, no -4.
- Siempre se cumple: `(a / b) * b + a % b` vale `a`.
- Por eso el resto toma el signo del primer operando.
- `%` solo funciona con enteros: con un `double` no compila.

% es de lo más útil

Expresión	Da	Sirve para
`n % 2`	0 si `n` es par, 1 si es impar y positivo	saber si es par
`n % 10`	último dígito de un `n` positivo	separar dígitos
`n / 10`	`n` sin su último dígito	separar dígitos
`total % 60`	segundos que sobran al formar minutos	convertir tiempos

Vienen ahora

El vuelto y el laboratorio se resuelven con `/` y `%`, sin ciclos.

Dividir entre cero

terminal

```
$ build/division_by_zero
Dos enteros: 5 0
division_by_zero.c:12:22: runtime error: division by zero
```

- Con enteros es **comportamiento indefinido**. El mensaje viene de `-fsanitize=undefined`.
- Lo que pasa después depende del procesador: se cae o sigue con un resultado sin sentido.
- Con `double`, en este entorno da `inf` y el programa sigue.

Evitarlo requiere preguntar si el divisor es cero: `if`, semana 4.

SECCIÓN 04

Conversiones

04

La trampa: `double half = 1 / 2;`

```
1 / 2
```

La trampa: `double half = 1 / 2;`



La trampa: `double half = 1 / 2;`



- La asignación es lo último que se ejecuta.
- Manda el tipo de los operandos, no el de la variable que recibe.

La trampa: `double half = 1 / 2;`



- La asignación es lo último que se ejecuta.
- Manda el tipo de los operandos, no el de la variable que recibe.
- Se corrige con un operando `double`: `1.0 / 2`.
- Entre `int` y `double`, el `int` se convierte a `double` antes de operar.

La trampa: `double half = 1 / 2;`



- La asignación es lo último que se ejecuta.
- Manda el tipo de los operandos, no el de la variable que recibe.
- Se corrige con un operando `double`: `1.0 / 2`.
- Entre `int` y `double`, el `int` se convierte a `double` antes de operar.
- `-Wall -Wextra` no avisa. Por eso se prueba con un ejemplo hecho a mano.

El cast

terminal

```
$ build/conversions
sum / count = 3.000000
(double) sum / count = 3.500000
(double) (sum / count) = 3.000000
from_int = 2.000000
truncated = 3
(int) 3.99 = 3
(int) -3.99 = -3
```

Con `sum` en 7 y `count` en 2.

Cast

`(tipo) valor` da ese valor convertido. La variable no cambia.

- Afecta solo a lo que tiene inmediatamente después.

El cast

terminal

```
$ build/conversions
sum / count = 3.000000
(double) sum / count = 3.500000
(double) (sum / count) = 3.000000
from_int = 2.000000
truncated = 3
(int) 3.99 = 3
(int) -3.99 = -3
```

Con `sum` en 7 y `count` en 2.

Cast

`(tipo) valor` da ese valor convertido. La variable no cambia.

- Afecta solo a lo que tiene inmediatamente después.
- `(double) (sum / count)` llega tarde: los decimales ya se perdieron.

El cast

terminal

```
$ build/conversions
sum / count = 3.000000
(double) sum / count = 3.500000
(double) (sum / count) = 3.000000
from_int = 2.000000
truncated = 3
(int) 3.99 = 3
(int) -3.99 = -3
```

Con `sum` en 7 y `count` en 2.

Cast

`(tipo) valor` da ese valor convertido. La variable no cambia.

- Afecta solo a lo que tiene inmediatamente después.
- `(double) (sum / count)` llega tarde: los decimales ya se perdieron.
- De `double` a `int`: los decimales se descartan, hacia cero. No redondea.

SECCIÓN 05

Entrada y salida

05

`printf` y sus formatos

Formato	Escribe	Ejemplo
`%d`	un `int`	`15`
`%f`	un `double`, con 6 decimales	`1.685000`
`%.2f`	un `double`, con 2 decimales	`1.69`
`%c`	un `char`	`B`
`%zu`	el resultado de `sizeof`	`4`
`%%`	el símbolo `%`	`%`

`printf` y sus formatos

Formato	Escribe	Ejemplo
`%d`	un `int`	`15`
`%f`	un `double`, con 6 decimales	`1.685000`
`%.2f`	un `double`, con 2 decimales	`1.69`
`%c`	un `char`	`B`
`%zu`	el resultado de `sizeof`	`4`
`%%`	el símbolo `%`	`%`

- Cada `%` es un hueco; los valores los llenan en orden.

`printf` y sus formatos

Formato	Escribe	Ejemplo
`%d`	un `int`	`15`
`%f`	un `double`, con 6 decimales	`1.685000`
`%.2f`	un `double`, con 2 decimales	`1.69`
`%c`	un `char`	`B`
`%zu`	el resultado de `sizeof`	`4`
`%%`	el símbolo `%`	`%`

- Cada `%` es un hueco; los valores los llenan en orden.
- `%.2f` solo cambia lo que se escribe, no la variable.

`printf` y sus formatos

Formato	Escribe	Ejemplo
`%d`	un `int`	`15`
`%f`	un `double`, con 6 decimales	`1.685000`
`%.2f`	un `double`, con 2 decimales	`1.69`
`%c`	un `char`	`B`
`%zu`	el resultado de `sizeof`	`4`
`%%`	el símbolo `%`	`%`

- Cada `%` es un hueco; los valores los llenan en orden.
- `%.2f` solo cambia lo que se escribe, no la variable.
- El formato debe coincidir con el valor: `%d` con un `double` es comportamiento indefinido. `-Wall` avisa.

`scanf`: leer un dato

read_grade.c

```c
int grade = 0;

printf("Nota: ");
scanf(" %d", &grade);
printf("Leí la nota %d\n", grade);
printf("Con dos puntos más sería %d\n", grade + 2);
```

terminal

```
$ build/read_grade
Nota: 15
Leí la nota 15
Con dos puntos más sería 17
```

- `" %d"`: qué esperar.

`scanf`: leer un dato

read_grade.c

```c
int grade = 0;

printf("Nota: ");
scanf("%d", &grade);
printf("Leí la nota %d\n", grade);
printf("Con dos puntos más sería %d\n", grade + 2);
```

terminal

```
$ build/read_grade
Nota: 15
Leí la nota 15
Con dos puntos más sería 17
```

- `"%d"`: qué esperar.
- `&grade`: **dónde** guardarlo. Es su dirección de memoria (semana 9).

`scanf`: leer un dato

read_grade.c

```c
int grade = 0;

printf("Nota: ");
scanf("%d", &grade);
printf("Leí la nota %d\n", grade);
printf("Con dos puntos más sería %d\n", grade + 2);
```

terminal

```
$ build/read_grade
Nota: 15
Leí la nota 15
Con dos puntos más sería 17
```

- `"%d"`: qué esperar.
- `&grade`: **dónde** guardarlo. Es su dirección de memoria (semana 9).

`scanf`: leer un dato

read_grade.c

```c
int grade = 0;

printf("Nota: ");
scanf("%d", &grade);
printf("Leí la nota %d\n", grade);
printf("Con dos puntos más sería %d\n", grade + 2);
```

terminal

```
$ build/read_grade
Nota: 15
Leí la nota 15
Con dos puntos más sería 17
```

- `"%d"`: qué esperar.
- `&grade`: **dónde** guardarlo. Es su dirección de memoria (semana 9).

	scanf	printf
int	%d	%d
double	%lf	%f
char	%c	%c

Dos errores que compilan: `-Wall` los avisa

terminal

```
$ gcc -Wall -Wextra -std=c17 broken/missing_ampersand.c -o build/missing_ampersand
broken/missing_ampersand.c:7:13: warning: format ‘%d’ expects argument of type ‘int *’, but argument 2 has type ‘int’ [-Wformat=]
    7 |     scanf("%d", grade);
      |           ~^   ~~~~~
      |            |   |
      |            |   int
      |            int *
```

terminal

```
$ gcc -Wall -Wextra -std=c17 broken/scanf_float_format.c -o build/scanf_float_format
broken/scanf_float_format.c:7:13: warning: format ‘%f’ expects argument of type ‘float *’, but argument 2 has type ‘double *’ [-Wformat=]
    7 |     scanf("%f", &price);
      |           ~^   ~~~~~~
      |            |   |
      |            |   double *
      |            float *
      |           %lf
```

%c lee cualquier carácter

char_trap.c

```c
printf("Cantidad: ");
scanf("%d", &quantity);
printf("Talla: ");
scanf("%c", &size);
```

terminal

```
$ build/char_trap
Cantidad: 3
Talla: Leí 3 y un carácter de código 10
```

- `%d` y `%lf` saltan espacios y saltos de línea.

%c lee cualquier carácter

char_trap.c

```c
printf("Cantidad: ");
scanf("%d", &quantity);
printf("Talla: ");
scanf("%c", &size);
```

terminal

```
$ build/char_trap
Cantidad: 3
Talla: Leí 3 y un carácter de código 10
```

- `%d` y `%lf` saltan espacios y saltos de línea.
- `%c` no: tomó el Enter, de código 10.

%c lee cualquier carácter

char_trap.c

```c
printf("Cantidad: ");
scanf("%d", &quantity);
printf("Talla: ");
scanf("%c", &size);
```

terminal

```
$ build/char_trap
Cantidad: 3
Talla: Leí 3 y un carácter de código 10
```

- `%d` y `%lf` saltan espacios y saltos de línea.
- `%c` no: tomó el Enter, de código 10.
- El programa no esperó la talla.

%c lee cualquier carácter

char_trap.c

```c
printf("Cantidad: ");
scanf("%d", &quantity);
printf("Talla: ");
scanf("%c", &size);
```

terminal

```
$ build/char_trap
Cantidad: 3
Talla: Leí 3 y un carácter de código 10
```

- `%d` y `%lf` saltan espacios y saltos de línea.
- `%c` no: tomó el Enter, de código 10.
- El programa no esperó la talla.

La solución

Un espacio antes: `scanf(" %c", &size);`

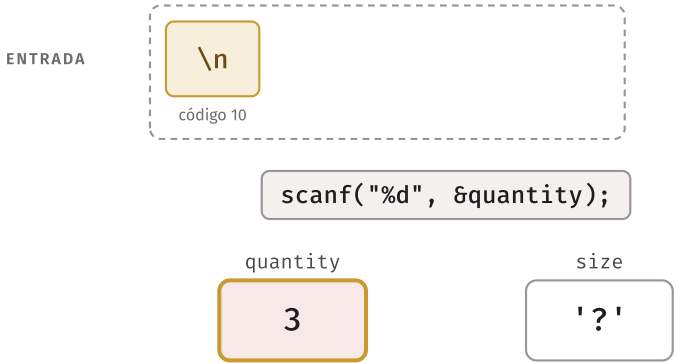
Lo que hay en la entrada





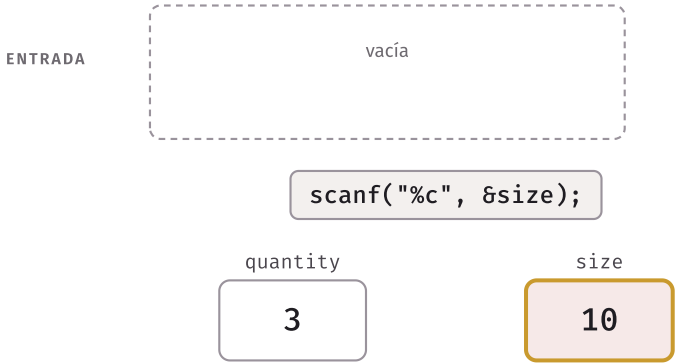
Se escribe 3 y Enter: a la entrada llegan dos caracteres

Lo que hay en la entrada



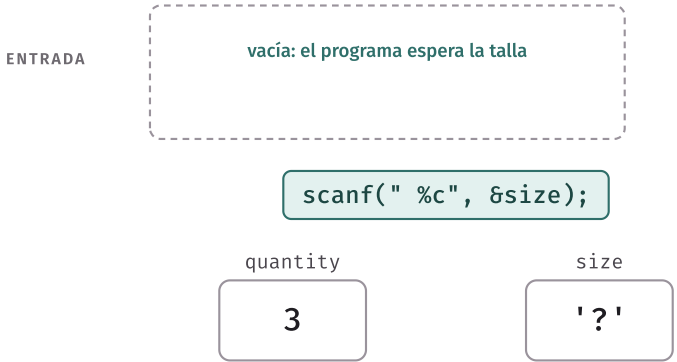
%d toma el 3 y deja ahí el salto de línea

Lo que hay en la entrada



%c toma el salto de línea, de código 10, y no espera la talla

Lo que hay en la entrada



En char_fixed.c, el espacio salta el salto de línea, y %c espera

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
4 9	2	guardó los dos

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
`4 9`	2	guardó los dos
`4 hola`	1	solo el primero

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
`4 9`	2	guardó los dos
`4 hola`	1	solo el primero
`3.7 5`	1	guardó 3; `.7` falla

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
`4 9`	2	guardó los dos
`4 hola`	1	solo el primero
`3.7 5`	1	guardó 3; `.7` falla
`hola`	0	nada

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
`4 9`	2	guardó los dos
`4 hola`	1	solo el primero
`3.7 5`	1	guardó 3; `.7` falla
`hola`	0	nada
`Ctrl+D`	-1	fin de la entrada: `EOF`

`scanf` devuelve cuántos datos guardó

read_check.c

```c
int count = scanf("%d %d", &first, &
    second);
```

terminal

```
$ build/read_check
Dos enteros: 4 hola
scanf devolvió 1
first = 4, second = 0
```

Se escribe	Da	Qué pasó
`4 9`	2	guardó los dos
`4 hola`	1	solo el primero
`3.7 5`	1	guardó 3; `.7` falla
`hola`	0	nada
`Ctrl+D`	-1	fin de la entrada: `EOF`

Lo que no se pudo leer se queda en la entrada, y la variable conserva su valor. Por eso la iniciamos.

La entrada puede fallar

terminal

```
$ build/igv_price
Precio sin IGV: hola
IGV: 0.00
Total: 0.00
```

El programa no avisa: escribe una cuenta falsa.

La entrada puede fallar

terminal

```
$ build/igv_price
Precio sin IGV: hola
IGV: 0.00
Total: 0.00
```

El programa no avisa: escribe una cuenta falsa.

- Un programa serio revisa lo que devuelve cada `scanf` y actúa.

La entrada puede fallar

terminal

```
$ build/igv_price
Precio sin IGV: hola
IGV: 0.00
Total: 0.00
```

El programa no avisa: escribe una cuenta falsa.

- Un programa serio revisa lo que devuelve cada `scanf` y actúa.
- Actuar requiere una decisión: el `if` de la semana 4.

La entrada puede fallar

terminal

```
$ build/igv_price
Precio sin IGV: hola
IGV: 0.00
Total: 0.00
```

El programa no avisa: escribe una cuenta falsa.

- Un programa serio revisa lo que devuelve cada `scanf` y actúa.
- Actuar requiere una decisión: el `if` de la semana 4.

Hasta entonces

Nuestros programas confían en que los datos llegan bien, y un comentario lo recuerda. No es que la entrada no pueda fallar: es que aún no tenemos cómo reaccionar.

SECCIÓN 06

De la fórmula al programa

06

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.
2. **Un ejemplo a mano**: elijan números y calculen. Es su prueba.

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.
2. **Un ejemplo a mano**: elijan números y calculen. Es su prueba.
3. **Pseudocódigo**.

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.
2. **Un ejemplo a mano**: elijan números y calculen. Es su prueba.
3. **Pseudocódigo**.
4. **C**, con cuidado con los tipos.

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.
2. **Un ejemplo a mano**: elijan números y calculen. Es su prueba.
3. **Pseudocódigo**.
4. **C**, con cuidado con los tipos.
5. **Comprobar**, con el ejemplo del paso 2 y con otros.

Cinco pasos

1. **Datos y resultado**: qué se lee, de qué tipo y qué se escribe.
2. **Un ejemplo a mano**: elijan números y calculen. Es su prueba.
3. **Pseudocódigo**.
4. **C**, con cuidado con los tipos.
5. **Comprobar**, con el ejemplo del paso 2 y con otros.

En pseudocódigo

`/` es la división exacta: `7 / 2` es 3.5. Para enteros usamos dos palabras: `7 div 2` es 3 y `7 mod 2` es 1.

Promedio final: $(EP + 2\,EF + PP)/4$

pseudocódigo

```
algoritmo promedio_final()
    leer ep
    leer ef
    leer pp
    promedio ← (ep + 2 * ef + pp) / 4
    escribir promedio
fin algoritmo
```

A mano, con 13, 15 y 14: $13 + 2 \cdot 15 + 14 = 57$, y $57/4 = 14.25$.

final_grade_bug.c

```
1   double final_grade = (ep + 2 * ef + pp) /
        4;
```

terminal

```
$ build/final_grade_bug
Examen parcial: 13
Examen final: 15
Promedio de prácticas: 14
Promedio final: 14.00
```

Promedio final: $(EP + 2\,EF + PP)/4$

pseudocódigo

```
algoritmo promedio_final()
    leer ep
    leer ef
    leer pp
    promedio ← (ep + 2 * ef + pp) / 4
    escribir promedio
fin algoritmo
```

A mano, con 13, 15 y 14: $13 + 2 \cdot 15 + 14 = 57$, y $57/4 = 14.25$.

final_grade_bug.c

```
1   double final_grade = (ep + 2 * ef + pp) /
        4;
```

terminal

```
$ build/final_grade_bug
Examen parcial: 13
Examen final: 15
Promedio de prácticas: 14
Promedio final: 14.00
```

Compila sin avisos, y es un bug.

El arreglo: `4.0`

final_grade.c

```c
int ep = 0;
int ef = 0;
int pp = 0;

printf("Examen parcial: ");
scanf("%d", &ep);
printf("Examen final: ");
scanf("%d", &ef);
printf("Promedio de prácticas: ");
scanf("%d", &pp);

double final_grade = (ep + 2 * ef + pp) / 4.0;

printf("Promedio final: %.2f\n", final_grade);
```

terminal

```
$ build/final_grade
Examen parcial: 13
Examen final: 15
Promedio de prácticas: 14
Promedio final: 14.25
```

El paso 2

Sin el ejemplo a mano, el bug pasa sin que nadie lo note.

El vuelto, ahora para cualquier monto

pseudocódigo

```
algoritmo vuelto(resto)
    monedas_500 ← resto div 500
    resto ← resto mod 500
    monedas_200 ← resto div 200
    resto ← resto mod 200
    monedas_100 ← resto div 100
    resto ← resto mod 100
    (igual con 50, 20 y 10)
    escribir las monedas y resto
fin algoritmo
```

- Se trabaja en **céntimos**, con un `int`: S/ 7.80 son 780.

change.c

```
1  int coins_500 = remaining / 500;
2  remaining = remaining % 500;
```

El vuelto, ahora para cualquier monto

pseudocódigo

```
algoritmo vuelto(resto)
    monedas_500 ← resto div 500
    resto ← resto mod 500
    monedas_200 ← resto div 200
    resto ← resto mod 200
    monedas_100 ← resto div 100
    resto ← resto mod 100
    (igual con 50, 20 y 10)
    escribir las monedas y resto
fin algoritmo
```

- Se trabaja en **céntimos**, con un `int`: S/ 7.80 son 780.
- `div`: cuántas monedas caben.
- `mod`: cuánto falta después.

change.c

```
1  int coins_500 = remaining / 500;
2  remaining = remaining % 500;
```

El vuelto, ahora para cualquier monto

pseudocódigo

```
algoritmo vuelto(resto)
    monedas_500 ← resto div 500
    resto ← resto mod 500
    monedas_200 ← resto div 200
    resto ← resto mod 200
    monedas_100 ← resto div 100
    resto ← resto mod 100
    (igual con 50, 20 y 10)
    escribir las monedas y resto
fin algoritmo
```

- Se trabaja en **céntimos**, con un `int`: S/ 7.80 son 780.
- `div`: cuántas monedas caben.
- `mod`: cuánto falta después.
- En C, `/` entre `int` ya es `div`, y `%` es `mod`.

change.c

```
1  int coins_500 = remaining / 500;
2  remaining = remaining % 500;
```

Siguiendo `remaining`

terminal

```
$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280

Siguiendo `remaining`

```
terminal

$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80

Siguiendo `remaining`

terminal

```
$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80
100	80	0	80

Siguiendo `remaining`

terminal

```
$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80
100	80	0	80
50	80	1	30

Siguiendo `remaining`

terminal

```
$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80
100	80	0	80
50	80	1	30
20	30	1	10

Siguiendo `remaining`

```
terminal

$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80
100	80	0	80
50	80	1	30
20	30	1	10
10	10	1	0

Siguiendo `remaining`

```
terminal

$ build/change
Vuelto en céntimos: 780
5 soles: 1
2 soles: 1
1 sol: 0
50 céntimos: 1
20 céntimos: 1
10 céntimos: 1
Sin cubrir: 0 céntimos
```

Moneda	Antes	Cuántas	Después
500	780	1	280
200	280	1	80
100	80	0	80
50	80	1	30
20	30	1	10
10	10	1	0

Son 5 monedas, como a mano. Con céntimos en un `int` la cuenta es exacta; la semana 3 explica por qué.

SECCIÓN 07

Biblioteca matemática y archivos

07

Funciones de `math.h`

Función	Calcula	Con 2.5
`sqrt(x)`	raíz cuadrada	`1.581139`
`pow(x, 3)`	x^3	`15.625000`
`fabs(x)`	valor absoluto	`2.500000`
`floor(x)`	hacia abajo	`2.000000`
`ceil(x)`	hacia arriba	`3.000000`
`round(x)`	más cercano	`3.000000`

- `#include <math.h>`
- `-lm` al final del `gcc`.
- Reciben y devuelven `double`.

`^` no es potencia

En C, `2 ^ 3` compila y hace otra cosa. Usen `pow`.

Si olvidan `-lm`

terminal

```
$ gcc -Wall -Wextra -std=c17 diagonal.c -o build/diagonal_no_lm
/usr/bin/ld: /tmp/ccFvYsrb.o: in function ‘main’:
diagonal.c:(.text+0x58): undefined reference to ‘sqrt’
collect2: error: ld returned 1 exit status
```

- El nombre `/tmp/ccFvYsrb.o` cambia en cada compilación; `+0x58` depende de la máquina.
- La traducción de su código salió bien. Falló el último paso: el **enlazador** (`ld`) no encontró `sqrt`.

La entrada y la salida pueden ser archivos

terminal

```
$ build/final_grade < grades.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
$ build/final_grade < grades.txt > build/result.txt
$ cat build/result.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
```

- `<` : la entrada viene de un archivo. `>` : la salida va a un archivo.

La entrada y la salida pueden ser archivos

terminal

```
$ build/final_grade < grades.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
$ build/final_grade < grades.txt > build/result.txt
$ cat build/result.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
```

- `<` : la entrada viene de un archivo. `>` : la salida va a un archivo.
- El programa no cambia. `grades.txt` tiene `13 15 14`, en una sola línea.

La entrada y la salida pueden ser archivos

terminal

```
$ build/final_grade < grades.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
$ build/final_grade < grades.txt > build/result.txt
$ cat build/result.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
```

- `<` : la entrada viene de un archivo. `>` : la salida va a un archivo.
- El programa no cambia. `grades.txt` tiene `13 15 14`, en una sola línea.
- Los mensajes también van al archivo: por eso el laboratorio no los usa.

La entrada y la salida pueden ser archivos

terminal

```
$ build/final_grade < grades.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
$ build/final_grade < grades.txt > build/result.txt
$ cat build/result.txt
Examen parcial: Examen final: Promedio de prácticas: Promedio final: 14.25
```

- `<` : la entrada viene de un archivo. `>` : la salida va a un archivo.
- El programa no cambia. `grades.txt` tiene `13 15 14`, en una sola línea.
- Los mensajes también van al archivo: por eso el laboratorio no los usa.
- `>` borra lo que había, sin preguntar.

Laboratorio: fórmulas en C

1. De Celsius a Fahrenheit con dos decimales. Cuidado con `9 / 5`.
2. Predecir y comprobar `7 / 2`, `7 % 2`, `7.0 / 2` y `(double) 7 / 2`.
3. De segundos a horas, minutos y segundos.
4. Datos en `entrada.txt`: `programa < entrada.txt > salida.txt`.
5. Bytes de cada tipo con `sizeof`.

Cómo trabajar

Empiecen desde `code/week-02/lab/`. `make test-lab` comprueba su trabajo, y falla hasta que lo terminen.

Para la próxima clase

- Leer los apuntes de la semana 3: bits, enteros, decimales y texto.
- Terminar el laboratorio.
- Hacer los ejercicios de los apuntes de esta semana.

Preguntas para pensar

¿Cuánto vale `2 + 3 * 4 % 5`? ¿Por qué `double p = 1 / 3;` guarda 0? ¿Por qué un `int` tiene un valor máximo, y cuál será?