

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Bits

Cómo se representan los datos

Gerson Garrido

Semana 3

Hoy

1. Todo son bits: binario y hexadecimal
2. Enteros con signo: complemento a dos
3. Overflow
4. Punto flotante: por qué `0.1 + 0.2` no da `0.3`
5. Texto: ASCII, Unicode y UTF-8
6. Git: guardar el historial de un trabajo

> **Lectura**
>
> Apuntes de la semana 3. El código está en `code/week-03`.

SECCIÓN 01

Todo son bits

01

Bit y byte

Bit

La unidad mínima de información: vale 0 o 1.

Byte

Un grupo de 8 bits. Cada byte de la memoria tiene su propio lugar.

Bits	Combinaciones
1	2

Bit y byte

Bit

La unidad mínima de información: vale 0 o 1.

Byte

Un grupo de 8 bits. Cada byte de la memoria tiene su propio lugar.

Bits	Combinaciones
1	2
3	8

Bit y byte

Bit

La unidad mínima de información: vale 0 o 1.

Byte

Un grupo de 8 bits. Cada byte de la memoria tiene su propio lugar.

Bits	Combinaciones
1	2
3	8
8	256

Bit y byte

Bit

La unidad mínima de información: vale 0 o 1.

Byte

Un grupo de 8 bits. Cada byte de la memoria tiene su propio lugar.

Bits	Combinaciones
1	2
3	8
8	256
32	4 294 967 296

Bit y byte

Bit

La unidad mínima de información: vale 0 o 1.

Byte

Un grupo de 8 bits. Cada byte de la memoria tiene su propio lugar.

Bits	Combinaciones
1	2
3	8
8	256
32	4 294 967 296
n	2^n

Los bits no traen etiqueta



como `int`: **65**

Idea clave

Un dato en memoria es solo un patrón de bits. Su significado lo pone el tipo con que se interpreta.

Los bits no traen etiqueta



como `int`: **65**

como `char`: **A**

Los bits no traen etiqueta



como `int`: **65**

como `char`: **A**

Idea clave

Un dato en memoria es solo un patrón de bits. Su significado lo pone el tipo con que se interpreta.

SECCIÓN 02

Binario y hexadecimal

02

Sistemas posicionales

Cada dígito vale según el lugar donde está.

$$202 = 2 \times 100 + 0 \times 10 + 2 \times 1$$

Sistemas posicionales

Cada dígito vale según el lugar donde está.

$$202 = 2 \times 100 + 0 \times 10 + 2 \times 1$$

Peso	128	64	32	16	8	4	2	1
Bit	1	1	0	0	1	0	1	0

Sistemas posicionales

Cada dígito vale según el lugar donde está.

$$202 = 2 \times 100 + 0 \times 10 + 2 \times 1$$

Peso	128	64	32	16	8	4	2	1
Bit	1	1	0	0	1	0	1	0

$$128 + 64 + 8 + 2 = \mathbf{202}$$

En base 2, cada posición vale el doble que la de su derecha.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

```
pseudocódigo

algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25
25	1	12

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25
25	1	12
12	0	6

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25
25	1	12
12	0	6
6	0	3

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25
25	1	12
12	0	6
6	0	3
3	1	1

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

De decimal a binario

pseudocódigo

```
algoritmo decimal_a_binario(n)
    mientras n ≠ 0 hacer
        escribir n mod 2
        n ← n div 2
    fin mientras
fin algoritmo
```

Los bits salen del de menor peso al de mayor peso: se leen de abajo hacia arriba.

`mod`: resto. `div`: cociente entero.

n	n mod 2	n div 2
202	0	101
101	1	50
50	0	25
25	1	12
12	0	6
6	0	3
3	1	1
1	1	0

Resultado: `11001010`

En C, todavía sin ciclos

to_binary.c (extracto)

```c
int n = 0;
int read_count = scanf("%d", &n);
printf("scanf leyó %d dato(s)\n",
     read_count);

int b0 = n % 2;
n = n / 2;
int b1 = n % 2;
n = n / 2;
// ... lo mismo para b2 a b6 ...
int b7 = n % 2;

printf("%d%d%d%d%d%d%d%d\n",
       b7, b6, b5, b4, b3, b2, b1, b0);
```

terminal

```
$ echo 202 | build/to_binary
scanf leyó 1 dato(s)
11001010
```

scanf puede fallar

Con `hola` escribe `scanf leyó 0 dato(s)`. Reaccionar a eso requiere `if`: la próxima semana.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5
1	11

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5
1	11
0	22

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5
1	11
0	22
1	45

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5
1	11
0	22
1	45
1	91

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

De binario a decimal

pseudocódigo

```
algoritmo binario_a_decimal(n)
    valor ← 0
    para i desde 0 hasta n - 1 hacer
        leer bit
        valor ← valor × 2 + bit
    fin para
    devolver valor
fin algoritmo
```

Bit	valor
1	1
0	2
1	5
1	11
0	22
1	45
1	91
0	182

En `from_binary.c`, `%1d` lee un solo dígito. El programa confirma el resultado de la tabla: 182, que en hexadecimal es `b6`.

Hexadecimal: 4 bits por dígito

Hex	Dec	Bin	Hex	Dec	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	a	10	1010
3	3	0011	b	11	1011
4	4	0100	c	12	1100
5	5	0101	d	13	1101
6	6	0110	e	14	1110
7	7	0111	f	15	1111

1100 **1010**

$$12 \times 16 + 10 = 202$$

Un byte son siempre dos dígitos hexadecimales. Por eso las herramientas que muestran bytes usan esta base.

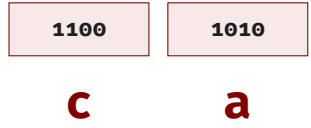
Hexadecimal: 4 bits por dígito

Hex	Dec	Bin	Hex	Dec	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	a	10	1010
3	3	0011	b	11	1011
4	4	0100	c	12	1100
5	5	0101	d	13	1101
6	6	0110	e	14	1110
7	7	0111	f	15	1111



Hexadecimal: 4 bits por dígito

Hex	Dec	Bin	Hex	Dec	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	a	10	1010
3	3	0011	b	11	1011
4	4	0100	c	12	1100
5	5	0101	d	13	1101
6	6	0110	e	14	1110
7	7	0111	f	15	1111



$$12 \times 16 + 10 = 202$$

Un byte son siempre dos dígitos hexadecimales. Por eso las herramientas que muestran bytes usan esta base.

Hexadecimal en C

bases.c (extracto)

```c
int n = 202;
int m = 0xCA;  // 0x indica que el literal está en
      hexadecimal

printf("%x en hexadecimal, con minúsculas\n", n);
printf("0xCA vale %d\n", m);
```

terminal

```
$ build/bases
202 en decimal
ca en hexadecimal, con minúsculas
CA en hexadecimal, con mayúsculas
0xCA vale 202
```

Error común

Creer que un número “está en hexadecimal” dentro de la variable.

La base es solo una forma de escribir

En memoria siempre hay bits. `n` y `m` guardan exactamente lo mismo.

Bytes y múltiplos

Decimal	Bytes	Binario	Bytes
1 kB	1000	1 KiB	1024
1 MB	1000^2	1 MiB	1024^2
1 GB	1000^3	1 GiB	1024^3

Bytes y múltiplos

Decimal	Bytes	Binario	Bytes
1 kB	1000	1 KiB	1024
1 MB	1000^2	1 MiB	1024^2
1 GB	1000^3	1 GiB	1024^3

- Un disco de 500 GB aparece como unos 465.66 GiB. Son los mismos bytes.

Bytes y múltiplos

Decimal	Bytes	Binario	Bytes
1 kB	1000	1 KiB	1024
1 MB	1000^2	1 MiB	1024^2
1 GB	1000^3	1 GiB	1024^3

- Un disco de 500 GB aparece como unos 465.66 GiB. Son los mismos bytes.
- **b** es bit y **B** es byte: un plan de 100 Mbps transfiere como máximo 12.5 MB por segundo.

SECCIÓN 03

Enteros con signo

03

Complemento a dos *(two's complement)*

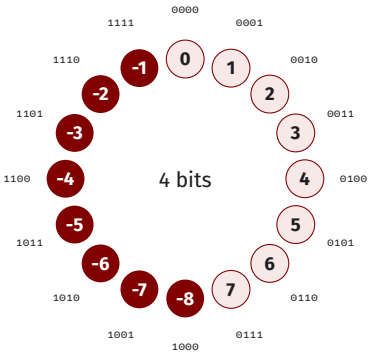
La idea

El bit de la izquierda tiene peso negativo.

-128	64	32	16	8	4	2	1
1	1	1	1	1	1	0	1

$$-128 + 64 + 32 + 16 + 8 + 4 + 1 = \mathbf{-3}$$

Con n bits, el rango va de -2^{n-1} a $2^{n-1} - 1$.



Cambiar el signo: invertir y sumar 1

3	`00000011`

Cambiar el signo: invertir y sumar 1

3	`00000011`
Invertir	`11111100`

Cambiar el signo: invertir y sumar 1

3	`00000011`
Invertir	`11111100`
Sumar 1	`11111101`

Cambiar el signo: invertir y sumar 1

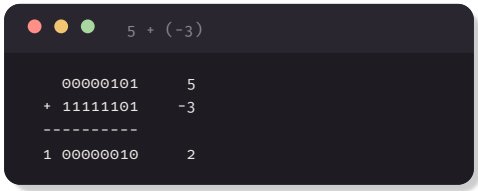
3	`00000011`
Invertir	`11111100`
Sumar 1	`11111101`

Por qué funciona: x más su versión invertida da `11111111`, que es -1.

Dos valores que conviene recordar

`11111111` es -1.
`10000000` es -128, y al cambiarle el signo vuelve a ser `10000000`: $+128$ no existe en 8 bits.

La suma es la misma



- El circuito no necesita saber si hay negativos.
- El `1` de la izquierda no cabe en 8 bits y se descarta.
- Restar se reduce a sumar el negativo.

El `int` en nuestro entorno

terminal

```
$ build/sizes
bits en un byte: 8
sizeof(char)   = 1
sizeof(int)    = 4
sizeof(double) = 8
INT_MAX = 2147483647
INT_MIN = -2147483648
```

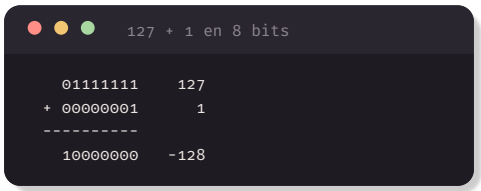
- Lo que dice el estándar: `sizeof(char)` es 1, y `int` tiene al menos 16 bits.
- Lo habitual hoy: 32 bits.
- $2^{31} - 1 = 2147483647$
- El gcc del entorno usa complemento a dos. C17 no lo exige; C23 sí.

SECCIÓN 04

Overflow

04

Un resultado que no cabe



Overflow

Un resultado que no cabe en el tipo que debe guardarlo.

Como un odómetro: después de `999999` viene `000000`.

Eso hace el hardware. C dice otra cosa.

En C, es comportamiento indefinido

Regla

El overflow de un entero con signo es **comportamiento indefinido** (undefined behavior). No es “da la vuelta”.

En C, es comportamiento indefinido

Regla

El overflow de un entero con signo es **comportamiento indefinido** (undefined behavior). No es “da la vuelta”.

- El estándar no dice qué debe pasar: no promete un resultado ni un error.

En C, es comportamiento indefinido

Regla

El overflow de un entero con signo es **comportamiento indefinido** (undefined behavior). No es “da la vuelta”.

- El estándar no dice qué debe pasar: no promete un resultado ni un error.
- El compilador puede suponer que nunca ocurre.

En C, es comportamiento indefinido

Regla

El overflow de un entero con signo es **comportamiento indefinido** (undefined behavior). No es “da la vuelta”.

- El estándar no dice qué debe pasar: no promete un resultado ni un error.
- El compilador puede suponer que nunca ocurre.
- El programa puede parecer correcto hoy y dar otro resultado con otras opciones.

En C, es comportamiento indefinido

Regla

El overflow de un entero con signo es **comportamiento indefinido** (undefined behavior). No es “da la vuelta”.

- El estándar no dice qué debe pasar: no promete un resultado ni un error.
- El compilador puede suponer que nunca ocurre.
- El programa puede parecer correcto hoy y dar otro resultado con otras opciones.

Sanitizer (UBSan)

Con `-fsanitize=undefined`, el programa revisa las operaciones mientras corre y avisa.

El mayor `int` más 1

overflow.c

```
6       int read_count = scanf("%d", &n);
7       printf("scanf leyó %d dato(s), n = %d\n", read_count, n);
8
9       int next = n + 1;
10      printf("n + 1 = %d\n", next);
```

El mayor `int` más 1

overflow.c

```
6       int read_count = scanf("%d", &n);
7       printf("scanf leyó %d dato(s), n = %d\n", read_count, n);
8
9       int next = n + 1;
10      printf("n + 1 = %d\n", next);
```

terminal

```
$ echo 41 | build/overflow
scanf leyó 1 dato(s), n = 41
n + 1 = 42
$ echo 2147483647 | build/overflow
scanf leyó 1 dato(s), n = 2147483647
overflow.c:9:9: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'
n + 1 = -2147483648
```

Cómo leer el mensaje

`overflow.c:9:9`	Archivo, línea y columna
`runtime error`	Error de ejecución, no de compilación
`signed integer overflow`	Overflow de un entero con signo
`cannot be represented in type 'int'`	La suma no cabe en un `int`

Error común

Ver `-2147483648` y concluir que "el `int` da la vuelta". Es lo que salió en esta ejecución, en esta máquina. C no lo garantiza. Sin el sanitizer, además, no hay ningún mensaje.

¿Por qué leer el valor con `scanf`?

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/constant_overflow.c -o build/constant_overflow
bad/constant_overflow.c: In function ‘main’:
bad/constant_overflow.c:6:24: warning: integer overflow in expression of type ‘int’ results in ‘-2147483648’ [-Woverflow]
    6 |     int next = INT_MAX + 1;
      |                        ^
```

- Si el valor está en el código, el compilador hace la cuenta y avisa al compilar.

¿Por qué leer el valor con `scanf`?

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/constant_overflow.c -o build/constant_overflow
bad/constant_overflow.c: In function ‘main’:
bad/constant_overflow.c:6:24: warning: integer overflow in expression of type ‘int’ results in ‘-2147483648’ [-Woverflow]
    6 |     int next = INT_MAX + 1;
      |                        ^
```

- Si el valor está en el código, el compilador hace la cuenta y avisa al compilar.
- UBSan trabaja en la ejecución: con `INT_MAX + 1` no da ningún mensaje.

¿Por qué leer el valor con `scanf`?

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/constant_overflow.c -o build/constant_overflow
bad/constant_overflow.c: In function ‘main’:
bad/constant_overflow.c:6:24: warning: integer overflow in expression of type ‘int’ results in ‘-2147483648’ [-Woverflow]
    6 |     int next = INT_MAX + 1;
      |                        ^
```

- Si el valor está en el código, el compilador hace la cuenta y avisa al compilar.
- UBSan trabaja en la ejecución: con `INT_MAX + 1` no da ningún mensaje.
- Para verlo, el valor tiene que llegar cuando el programa corre.

¿Por qué leer el valor con `scanf`?

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/constant_overflow.c -o build/constant_overflow
bad/constant_overflow.c: In function ‘main’:
bad/constant_overflow.c:6:24: warning: integer overflow in expression of type ‘int’ results in ‘-2147483648’ [-Woverflow]
    6 |     int next = INT_MAX + 1;
      |                        ^
```

- Si el valor está en el código, el compilador hace la cuenta y avisa al compilar.
- UBSan trabaja en la ejecución: con `INT_MAX + 1` no da ningún mensaje.
- Para verlo, el valor tiene que llegar cuando el programa corre.
- Con `if`, la próxima semana, podremos comprobar antes de sumar.

Sin signo sí da la vuelta

unsigned_wrap.c (extracto)

```c
unsigned char b = 255;
printf("b = %d\n", b);
b = b + 1;
printf("b = %d\n", b);
```

terminal

```
$ build/unsigned_wrap
b = 255
b = 0
```

C lo define

Los enteros sin signo trabajan módulo 2^n: no hay nada indefinido.

La suma se hace como `int` y da 256. Al guardarla en `unsigned char` se reduce módulo 256.

SECCIÓN 05

Punto flotante

05

0.1 no tiene escritura finita en binario

A la derecha del punto, los pesos son $\frac{1}{2}, \frac{1}{4}, \frac{1}{8} \ldots$

`0.11` en binario es $\frac{1}{2} + \frac{1}{4} = 0.75$: exacto.

Para 0.1: duplicar. La parte entera es el siguiente bit.

0.1 no tiene escritura finita en binario

A la derecha del punto, los pesos son $\frac{1}{2}, \frac{1}{4}, \frac{1}{8} \ldots$

`0.11` en binario es $\frac{1}{2} + \frac{1}{4} = 0.75$: exacto.

Para 0.1: duplicar. La parte entera es el siguiente bit.

x	$x \times 2$	**Bit**	Queda
0.1	0.2	0	0.2
0.2	0.4	0	0.4
0.4	0.8	0	0.8
0.8	1.6	1	0.6
0.6	1.2	1	0.2
0.2	0.4	0	0.4

Como 1/3 en decimal

`0.0001100110011...`
El grupo `0011` se repite sin fin.

Un `double`: notación científica en base 2



$$\text{número} = \pm 1.xxxxx_2 \times 2^{\text{exponente}}$$

Un `double`: notación científica en base 2



$$\text{número} = \pm 1.xxxxx_2 \times 2^{\text{exponente}}$$

- Conserva unas 15 a 17 cifras decimales significativas.

Un `double`: notación científica en base 2



$$\text{número} = \pm 1.xxxxx_2 \times 2^{\text{exponente}}$$

- Conserva unas 15 a 17 cifras decimales significativas.
- Solo algunos números son exactos: 0.5, 0.25 y 0.75 sí; 0.1, 0.2 y 0.3 no.

Un `double`: notación científica en base 2



$$\text{número} = \pm 1.xxxxx_2 \times 2^{\text{exponente}}$$

- Conserva unas 15 a 17 cifras decimales significativas.
- Solo algunos números son exactos: 0.5, 0.25 y 0.75 sí; 0.1, 0.2 y 0.3 no.
- Para esos se guarda el `double` más cercano.

0.1 + 0.2 con 20 decimales

terminal

```
$ build/float_sum
con %f:     0.300000
0.1       = 0.10000000000000000555
0.2       = 0.20000000000000001110
0.3       = 0.29999999999999998890
0.1 + 0.2 = 0.30000000000000004441
diferencia = 0.00000000000000005551
sum == c vale 0
```

1. 0.1 y 0.2 se guardan un poco por encima.

0.1 + 0.2 con 20 decimales

terminal

```
$ build/float_sum
con %f:      0.300000
0.1        = 0.10000000000000000555
0.2        = 0.20000000000000001110
0.3        = 0.29999999999999998890
0.1 + 0.2  = 0.30000000000000004441
diferencia = 0.00000000000000005551
sum == c vale 0
```

1. 0.1 y 0.2 se guardan un poco por encima.
2. Su suma cae en medio de dos `double`: se redondea al de arriba.

0.1 + 0.2 con 20 decimales

```
terminal

$ build/float_sum
con %f:      0.300000
0.1        = 0.10000000000000000555
0.2        = 0.20000000000000001110
0.3        = 0.29999999999999998890
0.1 + 0.2  = 0.30000000000000004441
diferencia = 0.00000000000000005551
sum == c vale 0
```

1. 0.1 y 0.2 se guardan un poco por encima.
2. Su suma cae en medio de dos `double`: se redondea al de arriba.
3. Ese no es el más cercano a 0.3: son dos valores vecinos.

0.1 + 0.2 con 20 decimales

```
terminal

$ build/float_sum
con %f:      0.300000
0.1        = 0.10000000000000000555
0.2        = 0.20000000000000001110
0.3        = 0.29999999999999998890
0.1 + 0.2  = 0.30000000000000004441
diferencia = 0.00000000000000005551
sum == c vale 0
```

1. 0.1 y 0.2 se guardan un poco por encima.
2. Su suma cae en medio de dos `double`: se redondea al de arriba.
3. Ese no es el más cercano a 0.3: son dos valores vecinos.
4. `%f` redondea a 6 decimales, y parece 0.3.

0.1 + 0.2 con 20 decimales

```
terminal

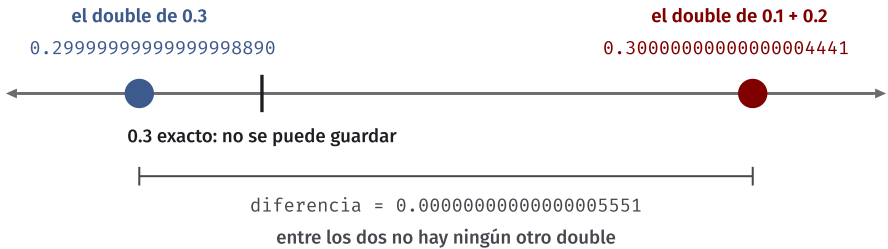
$ build/float_sum
con %f:      0.300000
0.1        = 0.10000000000000000555
0.2        = 0.20000000000000001110
0.3        = 0.29999999999999998890
0.1 + 0.2  = 0.30000000000000004441
diferencia = 0.00000000000000005551
sum == c vale 0
```

1. 0.1 y 0.2 se guardan un poco por encima.
2. Su suma cae en medio de dos `double`: se redondea al de arriba.
3. Ese no es el más cercano a 0.3: son dos valores vecinos.
4. `%f` redondea a 6 decimales, y parece 0.3.

Idea clave

No es un bug de C: es guardar en binario con una cantidad fija de bits.

Dos `double` vecinos



A escala. Para `==` son distintos, aunque ninguno de los dos sea 0.3.

Error común: comparar `double` con ==

compare_doubles.c (extracto)

```c
double sum = 0.1 + 0.2;
double target = 0.3;
double tolerance = 0.000000001;
printf("sum == target vale %d\n", sum == target);
printf("fabs(sum - target) < tolerance vale %d\n", fabs(sum - target) < tolerance);
```

terminal

```
$ build/compare_doubles
sum == target vale 0
fabs(sum - target) < tolerance vale 1
```

`==` compara y `=` asigna. Una comparación vale 1 si es verdadera y 0 si es falsa; la usaremos con `if`. La tolerancia se elige según el problema.

El dinero se guarda en céntimos

terminal

```
$ build/money
3 x 1.10 con double: 3.3000000000000026645
lo mismo con %.2f:   3.30
saldo con double:    0.0000000000000013878
3 x 110 céntimos:    330
en soles:            S/ 3.30
saldo en céntimos:   0
```

- El saldo empieza en un sol y se le restan diez veces 10 céntimos.

El dinero se guarda en céntimos

terminal

```
$ build/money
3 x 1.10 con double: 3.3000000000000026645
lo mismo con %.2f:   3.30
saldo con double:    0.0000000000000013878
3 x 110 céntimos:    330
en soles:            S/ 3.30
saldo en céntimos:   0
```

- El saldo empieza en un sol y se le restan diez veces 10 céntimos.
- Los enteros son exactos mientras no haya overflow.

El dinero se guarda en céntimos

terminal

```
$ build/money
3 x 1.10 con double: 3.3000000000000026645
lo mismo con %.2f:   3.30
saldo con double:    0.0000000000000013878
3 x 110 céntimos:    330
en soles:            S/ 3.30
saldo en céntimos:   0
```

- El saldo empieza en un sol y se le restan diez veces 10 céntimos.
- Los enteros son exactos mientras no haya overflow.
- Soles: `total / 100`. Céntimos: `total % 100`, con `%02d`.

El dinero se guarda en céntimos

terminal

```
$ build/money
3 x 1.10 con double: 3.30000000000000026645
lo mismo con %.2f:   3.30
saldo con double:    0.00000000000000013878
3 x 110 céntimos:    330
en soles:            S/ 3.30
saldo en céntimos:   0
```

- El saldo empieza en un sol y se le restan diez veces 10 céntimos.
- Los enteros son exactos mientras no haya overflow.
- Soles: `total / 100`. Céntimos: `total % 100`, con `%02d`.
- Límite con `int` de 32 bits: S/ 21 474 836.47.

SECCIÓN 06

Texto

06

`char` es un número pequeño: ASCII

Caracteres	Códigos
Salto de línea	10
Espacio	32
`'0'` a `'9'`	48 a 57
`'A'` a `'Z'`	65 a 90
`'a'` a `'z'`	97 a 122

ASCII

Tabla de 7 bits: 128 códigos, del 0 al 127.

Cada minúscula está 32 lugares después de su mayúscula.

Aritmética con caracteres

char_codes.c (extracto)

```c
char letter = 'A';
char digit = '7';
printf("%c tiene el código %d\n", letter, letter);
printf("%c tiene el código %d\n", letter + 1, letter + 1);
printf("'a' - 'A' = %d\n", 'a' - 'A');
printf("minúscula de %c: %c\n", letter, letter + 32);
printf("%c tiene el código %d\n", digit, digit);
printf("'7' - '0' = %d\n", digit - '0');
```

terminal

```
$ build/char_codes
A tiene el código 65
B tiene el código 66
'a' - 'A' = 32
minúscula de A: a
7 tiene el código 55
'7' - '0' = 7
```

Error común: confundir `'7'` con el número 7. El `char` guarda el código 55.

¿`char` tiene signo? Depende

gcc en ARM (Linux)

```
$ printf "ñ" | build/char_sign
scanf leyó 1 dato(s)
CHAR_MIN = 0, CHAR_MAX = 255
como char:          195
como unsigned char: 195
```

gcc con -fsigned-char

```
$ printf "ñ" | build/char_sign_signed
scanf leyó 1 dato(s)
CHAR_MIN = -128, CHAR_MAX = 127
como char:          -61
como unsigned char: 195
```

- C no lo dice: lo decide cada plataforma.
- En ARM con Linux, sin signo; en PC x86-64, con signo.
- Entre 0 y 127 no hay diferencia.

Consejo

Para el valor de un byte de 128 o más, usen `unsigned char`.

Carácter, código y byte

Concepto	Qué es	Ejemplo
Carácter	La letra o el símbolo, como idea	`ñ`
Código	El número que le asigna Unicode	`U+00F1`, que es 241
Bytes	Cómo se guarda ese número	`c3 b1` en UTF-8

- ASCII no incluye la `ñ`. Antes, cada región usaba su propia tabla de un byte.

Carácter, código y byte

Concepto	Qué es	Ejemplo
Carácter	La letra o el símbolo, como idea	`ñ`
Código	El número que le asigna Unicode	`U+00F1`, que es 241
Bytes	Cómo se guarda ese número	`c3 b1` en UTF-8

- ASCII no incluye la `ñ`. Antes, cada región usaba su propia tabla de un byte.
- **Unicode**: una sola tabla para todos los sistemas de escritura.

Carácter, código y byte

Concepto	Qué es	Ejemplo
Carácter	La letra o el símbolo, como idea	`ñ`
Código	El número que le asigna Unicode	`U+00F1`, que es 241
Bytes	Cómo se guarda ese número	`c3 b1` en UTF-8

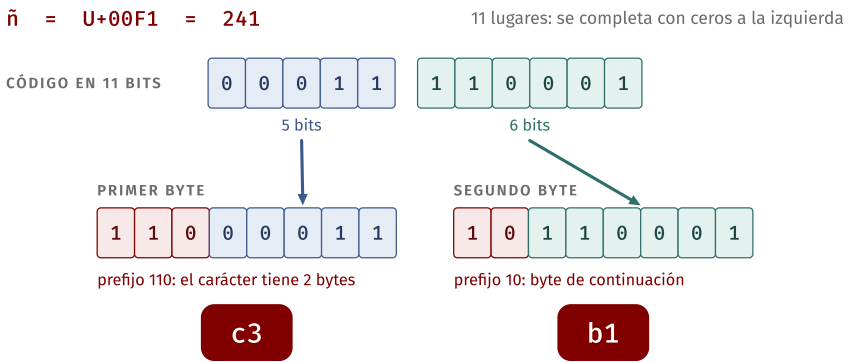
- ASCII no incluye la `ñ`. Antes, cada región usaba su propia tabla de un byte.
- **Unicode**: una sola tabla para todos los sistemas de escritura.
- **UTF-8**: la codificación más usada. Todo texto ASCII ya es UTF-8 válido.

Los patrones de UTF-8

Rango	Decimal	Bytes	Patrón
`U+0000` a `U+007F`	0 a 127	1	`0xxxxxxx`
`U+0080` a `U+07FF`	128 a 2047	2	`110xxxxx 10xxxxxx`
`U+0800` a `U+FFFF`	2048 a 65535	3	`1110xxxx 10xxxxxx 10xxxxxx`
`U+10000` a `U+10FFFF`		4	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Las `x` son los bits del código. El primer byte dice cuántos bytes tiene el carácter; los demás empiezan con `10`.

La ñ, a mano



Por eso la `ñ` ocupa 2 bytes. La `n` (código 110, en el rango de ASCII) ocupa 1 byte: `6e`.

Mirar los bytes con `xxd`

terminal

```
$ echo "año" > year.txt
$ xxd year.txt
00000000: 61c3 b16f 0a                            a..o.
$ wc -c year.txt
5 year.txt
```

61	c3 b1	6f	0a
a	ñ	o	salto de línea

3 caracteres visibles y 5 bytes en total. El `0a` lo agrega `echo`.

Caracteres y bytes no son lo mismo

Carácter	Código	Bytes
n	U+006E	1
ñ	U+00F1	2
euro	U+20AC	3
un emoji	U+1F600	4

Idea clave

Número de caracteres y número de bytes no son lo mismo. Con ñ y tildes deja de coincidir.

terminal

```
$ echo "ñandú" | xxd
00000000: c3b1 616e 64c3 ba0a                      ..and...
```

ñandú: 5 caracteres y 7 bytes, sin contar el salto de línea.

La tabla equivocada

terminal

```
$ iconv -f latin1 -t utf8 year.txt
aÃ±o
```

- El archivo está bien. Le dijimos a `iconv` que está en Latin-1, y no es cierto.

La tabla equivocada

terminal

```
$ iconv -f latin1 -t utf8 year.txt
aÃ±o
```

- El archivo está bien. Le dijimos a `iconv` que está en Latin-1, y no es cierto.
- En Latin-1, `c3` es Ã y `b1` es ±.

La tabla equivocada

terminal

```
$ iconv -f latin1 -t utf8 year.txt
aÃ±o
```

- El archivo está bien. Le dijimos a `iconv` que está en Latin-1, y no es cierto.
- En Latin-1, `c3` es Ã y `b1` es ±.
- Seguramente lo han visto en algún correo o página web.

La ñ no cabe en un `char`

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/char_enye.c -o build/char_enye
bad/char_enye.c: In function ‘main’:
bad/char_enye.c:5:14: warning: multi-character character constant [-Wmultichar]
    5 |     char c = ’ñ’;
      |              ^~~
bad/char_enye.c:5:14: warning: unsigned conversion from ‘int’ to ‘char’ changes value from ‘50097’ to ‘177’ [-Woverflow]
```

- Hay dos bytes entre las comillas simples.

La ñ no cabe en un `char`

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/char_enye.c -o build/char_enye
bad/char_enye.c: In function ‘main’:
bad/char_enye.c:5:14: warning: multi-character character constant [-Wmultichar]
    5 |     char c = ’ñ’;
      |              ^~~
bad/char_enye.c:5:14: warning: unsigned conversion from ‘int’ to ‘char’ changes value from ‘50097’ to ‘177’ [-Woverflow]
```

- Hay dos bytes entre las comillas simples.
- 50097 es `c3b1`. Queda solo `b1`, que no es un carácter válido por sí solo.

La ñ no cabe en un `char`

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/char_enye.c -o build/char_enye
bad/char_enye.c: In function ‘main’:
bad/char_enye.c:5:14: warning: multi-character character constant [-Wmultichar]
    5 |     char c = ’ñ’;
      |              ^~~
bad/char_enye.c:5:14: warning: unsigned conversion from ‘int’ to ‘char’ changes value from ‘50097’ to ‘177’ [-Woverflow]
```

- Hay dos bytes entre las comillas simples.
- 50097 es `c3b1`. Queda solo `b1`, que no es un carácter válido por sí solo.
- La segunda advertencia depende de la plataforma.

La ñ no cabe en un `char`

terminal

```
$ gcc -Wall -Wextra -std=c17 bad/char_enye.c -o build/char_enye
bad/char_enye.c: In function ‘main’:
bad/char_enye.c:5:14: warning: multi-character character constant [-Wmultichar]
    5 |     char c = 'ñ';
      |              ^~~
bad/char_enye.c:5:14: warning: unsigned conversion from ‘int’ to ‘char’ changes value from ‘50097’ to ‘177’ [-Woverflow]
```

- Hay dos bytes entre las comillas simples.
- 50097 es `c3b1`. Queda solo `b1`, que no es un carácter válido por sí solo.
- La segunda advertencia depende de la plataforma.
- `%c` lee un byte, no un carácter.

SECCIÓN 07

Git

07



Para qué sirve Git

`tarea_final.c`, `tarea_final2.c`, `tarea_final_ahora_si.c`...

Repositorio

Una carpeta cuyo historial lleva Git, guardado en la subcarpeta oculta `.git`.

Commit

Un registro de cómo estaban los archivos, con un mensaje que dice qué cambió.

En el laboratorio

Creen el repositorio de práctica en ~, no dentro de la carpeta del curso.

Dos pasos: preparar y registrar



- `git add` elige qué cambios entran en el próximo commit. Todavía no guarda nada en el historial.
- `git commit -m "mensaje"` crea el commit.
- `git status` muestra en qué estado está cada archivo.

Crear el repositorio y el primer archivo

terminal

```
$ mkdir lab03-git
$ cd lab03-git
$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
[...]
Initialized empty Git repository in /root/lab03-git/.git/
$ echo "Práctica 1: 15" > grades.txt
$ git add grades.txt
```

Aquí se omiten varias líneas `hint:` (`[...]`): son sugerencias sobre branches, y por ahora se pueden ignorar. La ruta de la última línea depende del entorno.

El primer commit

terminal

```
$ git commit -m "Agrega la nota de la práctica 1"
Author identity unknown

*** Please tell me who you are.
[...]
fatal: unable to auto-detect email address (got 'root@616988be0ce2.(none)')
$ git config --global user.name "Ana Quispe"
$ git config --global user.email "ana@example.com"
$ git commit -m "Agrega la nota de la práctica 1"
[master (root-commit) 3517bfb] Agrega la nota de la práctica 1
 1 file changed, 1 insertion(+)
 create mode 100644 grades.txt
```

`[...]` indica líneas omitidas: el mensaje completo está en los apuntes. Se configura una vez, con sus datos. En algunos entornos ya viene configurado. El `3517bfb` será otro en su máquina.

`git diff`: lo que cambió

terminal

```
$ echo "Práctica 2: 17" >> grades.txt
$ git diff
diff --git a/grades.txt b/grades.txt
index 12e499b..c4a572a 100644
--- a/grades.txt
+++ b/grades.txt
@@ -1 +1,2 @@
 Práctica 1: 15
+Práctica 2: 17
```

- » agrega al final; > reemplaza.
- La línea con `+` es nueva.
- Conviene leer el `diff` antes del commit.

Error común

Sin `git add`, el cambio no entra en el commit.

`git log`: el historial

terminal

```
$ git add grades.txt
$ git commit -m "Agrega la nota de la práctica 2"
[master f7f7370] Agrega la nota de la práctica 2
 1 file changed, 1 insertion(+)
$ git log --oneline
f7f7370 Agrega la nota de la práctica 2
3517bfb Agrega la nota de la práctica 1
```

- El commit más reciente aparece primero. Los identificadores cambian en cada máquina.
- Si `git log` no cabe en la pantalla, se sale con la tecla `q`.
- Un buen mensaje dice qué cambió: no “cambios”, sino “Agrega la nota de la práctica 2”.

Laboratorio: bits a mano y Git

1. Convertir 202 a binario y a hexadecimal, y comprobarlo con `%x`.
2. Sumar 1 al mayor `int`, leído con `scanf`, y leer el mensaje del sanitizer.
3. Escribir `0.1 + 0.2` con `%.20f`.
4. Crear `year.txt` con `año` y encontrar los bytes de la `ñ` con `xxd`.
5. Crear un repositorio, hacer dos commits y revisar `status`, `diff` y `log`.

Cómo se comprueba

Dentro de `code/week-03`, `make test-lab` indica qué falta. Git se revisa en el laboratorio.

Para la próxima clase

- Leer los apuntes de la semana 3 y resolver sus ejercicios.
- La próxima semana: decisiones con `if`. Por fin podremos reaccionar cuando `scanf` falla.
- Práctica 1 en la semana 4.

Pregunta

Un archivo tiene 5 caracteres y 7 bytes en UTF-8. ¿Cómo es posible? ¿Y cuántos bytes tendría si todos sus caracteres fueran ASCII?