# Sesión 03: Tipos básicos

Programación para Sistemas

Ángel Herranz

Otoño 2018

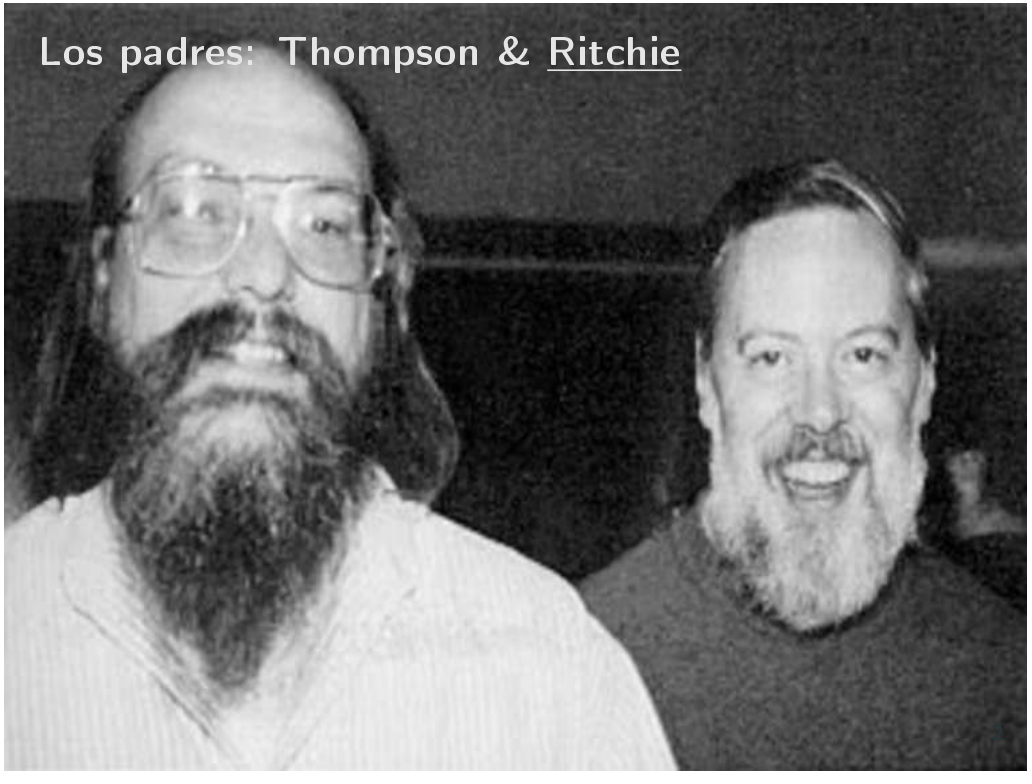
Universidad Politécnica de Madrid

## Recordatorio

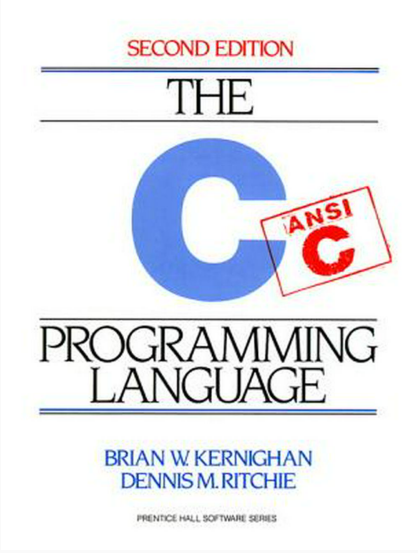
- ¿Cómo van esas instalaciones de Ubuntu?
- ¿Cómo van esos accesos a `triqui`?
- ¿Cómo va Bash?
- ¿Y C?
- ¿Cómo va el repaso de las transparencias?
- ¿Cómo van las hojas de ejercicios?



## Los padres: Thompson & Ritchie



## Cómprate este libro



## ¿Qué nos hace ingenieros?

Las matemáticas

Lenguaje = Sintaxis + Semántica

::= ⟦⟧



## ¿Por donde empiezo?

Por los tipos

Una semántica útil y sencilla:

un tipo es un conjunto de datos



## Tipos básicos en C (C es realmente pequeño)

| | |
|---|---|
| **char** | los enteros que quepan en 1 byte |
| **int** | los enteros que quepan en la palabra de la máquina |
| **float** | coma flotante *simple* |
| **double** | coma flotante *doble*[1] |

[1] Tanto **float** como **double** siguen el estándar de coma flotante IEEE 754.



## Exploremos esos tipos i

Transcribir, compilar y ejecutar el siguiente programa:

```
1  #include <stdio.h>
2  int main() {
3    char mi_char = 'a';
4    int mi_int = 42;
5    float mi_float = 1000000.0;
6    double mi_double = 0.0000001;
7    printf("El char es: %c\n", mi_char);
8    printf("El int es es: %d\n", mi_int);
9    printf("El float es es: %f\n", mi_float);
10   printf("El double es: %F\n", mi_double);
11   return 0;
12 }
```

## Exploremos esos tipos ii

```
El char es: a
El int es: 42
El float es: 1000000.000000
El double es: 0.000000
```



## printf: conversión % (del libro K&R)

Conviene tener esta tabla muy a mano:

TABLE 7-1. BASIC PRINTF CONVERSIONS

| CHARACTER | ARGUMENT TYPE; PRINTED AS |
|---|---|
| d, i | int; decimal number. |
| o | int; unsigned octal number (without a leading zero). |
| x, X | int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15. |
| u | int; unsigned decimal number. |
| c | int; single character. |
| s | char *; print characters from the string until a '\0' or the number of characters given by the precision. |
| f | double; [−]*m.dddddd*, where the number of *d*'s is given by the precision (default 6). |
| e, E | double; [−]*m.dddddd* e±*xx* or [−]*m.dddddd* E±*xx*, where the number of *d*'s is given by the precision (default 6). |
| g, G | double; use %e or %E if the exponent is less than −4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed. |
| p | void *; pointer (implementation-dependent representation). |
| % | no argument is converted; print a %. |



## Exploremos esos tipos iii

```
5   float mi_double = 0.0000001;

    El double es: 0.000000

10  printf("El double es: %.7F\n", mi_double);

    El double es: 0.0000001

    printf("El double es: %.23F\n", mi_double);

    El double es: 0.00000010000000000000000

    printf("El double es: %.24F\n", mi_double);

    El double es: 0.000000099999999999999995
```



Una cosa es lo que hay en una variable y otra lo que el `printf` y equivalentes exponen.

Esta afirmación es válida para cualquier lenguaje de programación.

## Siempre tenemos el manual de `stdio`

Desde Bash:

`man 3 printf`



## Algunas características de los enteros

- No hay booleanos
- Se usan enteros en las condiciones de **`if`**s y bucles:

  Igual a 0 ≡ false

  Distinto de 0 ≡ true
- **`char`** e **`int`** son enteros

Explicar:

```
7  printf("El char es: %d\n", mi_char);
8  printf("El int es es: %c\n", mi_int);
```

```
El char es: a
El int es es: 42
```

vs.

```
El char es: 97
El int es es: *
```



## Exploremos esos tipos iv

**`sizeof`**

- Operador predefinido (no es una función de biblioteca)
- Admite tipos:

  **`sizeof(int)`**
- Y expresiones:

  **`sizeof`**`(5 + mi_int)`
- Devuelve el tamaño del argumento en bytes

Modificar el programa anterior para que imprima el tamaño de las variables `mi_char`, `mi_int`, `mi_float`, `mi_double`.

Escribir un programa que imprima el tamaño de los tipos básicos.



## Antes de continuar

- ¿Qué hacemos con esto?

```
sizeof.c: In function ‘main’:
sizeof.c:7:36: warning: format ‘%d’ expects argument of
   type ‘int’, but argument 2 has type
   ‘long unsigned int’ [-Wformat=]

   printf("El tamano del char es: %d\n", sizeof(mi_char));
                                        ~^
                                        %ld
```

- El compilador nos está diciendo que **`sizeof`** no devuelve **`int`** si no **`long unsigned int`**
- De momento, podemos corregirlo usando la conversión `%ld` en vez de `%d` (“`l`” indica **`long`**)

## Pidamos ayuda al compilador

A partir de ahora usaremos los siguientes *flags*:

```
gcc -ansi -Wall -Werror -pedantic ...
```

- `-ansi` *This turns off certain features of GCC that are incompatible with ISO C90 . . .*
- `-Wall` *This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid . . .*
- `-Werror` *Make all warnings into errors.*

  No genera ejecutable cuando aún hay warnings
- `-pedantic` *Issue all the warnings demanded by strict ISO C . . .*



## El significado en el tipo char

$$[\![\texttt{expresión en C}]\!] = \text{significado matemático}$$

$$[\![\texttt{char}]\!] = \{-128, -127, \ldots, -2, -1, 0, 1, 2, \ldots, 127\}$$
$$[\![\texttt{97}]\!] = 97$$
$$[\![\texttt{'a'}]\!] = 97$$
$$[\![\texttt{mi\_char}]\!] = 97$$
$$[\![\texttt{'a'== 97}]\!] = \; ?$$



## *Overflows* i

```
char c = 'a';
c = c + c;
printf("%d\n",c);
```

$$[\![\texttt{c}]\!] = -62$$

¿Por qué?



## Necesitamos bajar a nivel máquina

| | |
|---|---|
| 0 0 0 0 0 0 0 0 | 0 |
| 0 0 0 0 0 0 0 1 | 1 |
| 0 0 0 0 0 0 1 0 | 2 |
| 0 0 0 0 0 0 1 1 | 3 |
| ... | |
| 0 1 1 1 1 1 1 0 | 126 |
| 0 1 1 1 1 1 1 1 | 127 |

| | |
|---|---|
| 1 0 0 0 0 0 0 0 | −128 |
| 1 0 0 0 0 0 0 1 | −127 |
| 1 0 0 0 0 0 1 0 | −126 |
| 1 0 0 0 0 0 1 1 | −125 |
| ... | |
| 1 1 1 1 1 1 1 0 | −2 |
| 1 1 1 1 1 1 1 1 | −1 |

Complemento a 2

## El significado en todos los enteros

Sólo cambia el número de bytes:

8, 32, 64, 128, ...



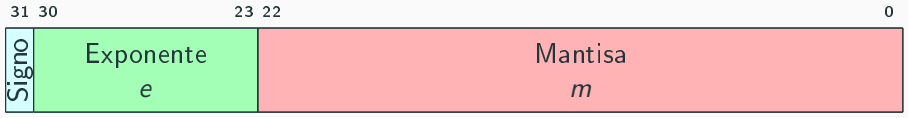
## *Overflows* ii

Cuidado con los *overflows*

Pasa con todos los tipos básicos.



## Retomemos la coma flotante por un momento




$$(-1)^{b_{31}} \times (1 + \sum_{i=1}^{23} b_{23-i} \times 2^{-i}) \times 2^{e-127}$$

(IEEE 754 binary32)

- ¡Sólo fracciones binarias! $\Longrightarrow$ Pérdida de precisión
- Aproximadamente, ¿qué número es este **float**?

| 0 | 0 1 1 1 1 1 0 1 | 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 |
|---|---|---|



## Pérdida de precisión en la práctica

```
float a = 0.0001;
float b = 0.0003;
float f1 = a / b;
a = 1.0;
b = 3.0;
float f2 = a / b;
if (f1 == f2) {
  printf("iguales\n");
}
else {
  printf("desiguales\n");
}
```

- Transcribir y ejecutar
- desiguales
  - No pasa solo en C.
  - Nunca puede confiarse en las comparaciones.
  - Se usan trucos del tipo:

$$|f_1 - f_2| < \epsilon$$

  - O en código:

```
fabs(f1 - f2) < 0.001
```

## Lectura

*What every computer scientist should know about floating-point arithmetic*

David Goldberg



## *Qualifiers* i

# `unsigned`

- Convierte un tipo entero en un tipo *natural*:

sólo positivos

- Por ejemplo:

$$[\![\texttt{unsigned char}]\!] = \{0, 1, 2, \ldots, 255\}$$

- Igual para todos los tipos enteros.



## ¿Todos los caracteres?

- Escribir y ejecutar un programa que imprima todos los caracteres de 0 al 255.



## *Qualifiers* ii

# `long`

- Aumenta el tamaño de los tipos **`int`** y **`double`**:

más valores, más precisión

- Se pueden poner dos **`long`**s:

```
long long int
```

- Se puede combinar con **`unsigned`**

```
unsigned long int
unsigned long long int
```

## Parte de la *Standard Library*: `limits.h`

- Define constantes para el tamaño de los tipos entero.
- Basta con incluir el *header*: **`#include`** `<limits.h>`
- Y ya se pueden usar constantes como `CHAR_MIN`, `CHAR_MAX`, `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`, `UINT_MIN`, `UINT_MAX`, ...
- Busca en tu máquina el *header* `limits.h` y explóralo con un editor de texto.
- Otra parte de la biblioteca: `float.h`
- Busca en tu máquina el *header* `float.h` y explóralo con un editor de texto.



# Gramáticas: *crash course*

## Gramáticas

- Vais a ser ingenieros, las matemáticas son vuestra herramienta principal
- En CS (*Computer Science*) tenemos un montón de matemáticas a nuestro servicio
- Entre esas herramientas: *teoría de lenguajes*
- Para saber lo que podemos escribir en un lenguaje necesitamos una gramática
- Vemos en este *crash course* un ejemplo de un lenguaje inventado
- Usaremos gramáticas para ver cómo se escriben ciertas frases en C



## Un mini lenguaje de programación i

⟨*stm*⟩ ::= <assign>
| <if>
| <while>
| <block>

⟨*assign*⟩ ::= <lvalue> ':=' <exp>

⟨*lvalue*⟩ ::= <var>

⟨*if*⟩ ::= 'if' '(' <exp> ')' <stm>

⟨*while*⟩ ::= 'while' '(' <exp> ')' <stm>

⟨*block*⟩ ::= 'begin' <stm_list> 'end'

⟨*stm_list*⟩ ::= <stm> ';'
| <stm> ';' <stm_list>

## Un mini lenguaje de programación ii

$\langle exp \rangle$ ::= <cte>
| <var>
| <exp> '=' <exp>
| <exp> '+' <exp>

$\langle cte \rangle$ ::= '0' | '-1' | '1' | '-2' | '2' | ... | '32767' | '-32768'

$\langle var \rangle$ ::= 'a' | 'b' | ... | 'z'



## Un mini lenguaje de programación iii

- Escribir frases del lenguaje (empieza por las reglas sin recursión, mejor las constantes)