

Sesión 03: Tipos básicos

Programación para Sistemas

Ángel Herranz

2022-2023

Universidad Politécnica de Madrid

Recordatorio

- ¿Cómo van esas instalaciones de Ubuntu?
- ¿Cómo va el repaso de las transparencias?
- ¿Cómo van las hojas de ejercicios?
- ¿Cómo va Bash? (`pwd`, `ls`, `cd`, `mkdir`, ...)
- ¿Cómo van esos accesos a triqui? (`ssh`)
- ¿Y C y el depurador? (`gcc -g`, `gdb ./a.out`)



En el capítulo de hoy empezamos con...

THE C PROGRAMMING LANGUAGE

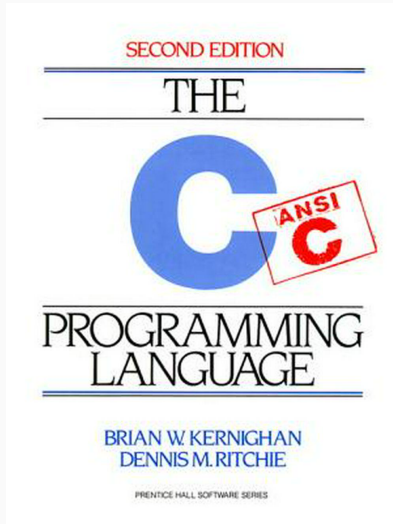
Thompson & Ritchie en *Bell Labs*



Antes de comprar el último baile del *Fortnite*...



... cómprate este libro



Para aprender un lenguaje

Sintaxis + Semántica

Sintaxis + Semántica



Para aprender un lenguaje

Sintaxis + Semántica



¿Por donde empiezo?

¿Por donde empiezo?

Por los tipos

¿Por donde empiezo?

Por los tipos

Una semántica útil y sencilla:

un tipo es el conjunto de datos que
puede haber en una variable de ese
tipo

¿Por donde empiezo?

Por los tipos

Una semántica útil y sencilla:

un tipo es el conjunto de datos que
puede haber en una variable de ese
tipo

(un poco pobre pero nos vale)

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 <i>simple</i>
double	coma flotante <i>doble</i> ¹

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

²Sintaxis simplificada

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 <i>simple</i>
double	coma flotante <i>doble</i> ¹

Declaración de variable (similar a Java)²:

$\langle \textit{declaration} \rangle ::= \langle \textit{type_specifier} \rangle \langle \textit{identifier} \rangle ' ; '$

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

²Sintaxis simplificada

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 <i>simple</i>
double	coma flotante <i>doble</i> ¹

Declaración de variable (similar a Java)²:

$\langle \textit{declaration} \rangle ::= \langle \textit{type_specifier} \rangle \langle \textit{identifier} \rangle ' ; '$
char mi_char;

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

²Sintaxis simplificada

Exploreemos esos tipos i

 Transcribir, compilar y ejecutar el programa `basicos.c`:

```
#include <stdio.h>
int main() {
    char mi_char = 'a';
    int mi_int = 42;
    float mi_float = 1000000.0;
    double mi_double = 0.0000001;
    printf("El char es: %c\n", mi_char);
    printf("El int es es: %d\n", mi_int);
    printf("El float es es: %f\n", mi_float);
    printf("El double es: %f\n", mi_double);
    return 0;
}
```

¿Comentarios?

¿Comentarios?

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.ddddd, where the number of d's is given by the precision (default 6).
e, E	double; [-]m.ddddd e±xx or [-]m.ddddd 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

```
float mi_double = 0.0000001;  
printf("El double es: %f\n", mi_double);
```

El double es: 0.000000

Exploremos esos tipos iii

```
float mi_double = 0.0000001;  
printf("El double es: %f\n", mi_double);
```

El double es: 0.000000

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

El double es: 0.0000001

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

El double es: 0.000000100000000000000000

```
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.

Desde Bash:

man 3 printf

Algunas características de los enteros i

- En C no hay booleanos
- **char** e **int** son enteros
- Se usan enteros en las condiciones de **ifs** y bucles:

Igual a 0 $\equiv$ false

Distinto de 0 $\equiv$ true

- Eso y algunas otras *locuras* de C nos permiten escribir cosas de forma muy concisa:

```
while (--n) f *= n;
```

 ¿Qué hace eso?

 5'

Algunas características de los enteros ii

 ¿Puedes explicar el resultado del siguiente cambio?

```
3  char mi_char = 'a';  
4  int mi_int = 42;
```

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

El char es: a
El int es: 42

vs.

El char es: 97
El int es: *

Exploremos esos tipos iv

sizeof

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

sizeof(int)

- Y expresiones:

sizeof(mi_int)

sizeof(mi_int + 5)

- Devuelve el tamaño del argumento en bytes

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

 5'

Exploremos esos tipos v

-  Escribir un programa (`sizeof.c`) que imprima el tamaño de los tipos básicos

 5'

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 tamaño 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 ...*

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 ...*

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

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 ...*

Make

```
$ gcc -ansi -Wall -Werror -pedantic -o basics basics.c
```

Make

```
$ gcc -ansi -Wall -Werror -pedantic -o basics basics.c
```

- ¿Cada vez que quiero compilar?
- ¿De verdad que no hay un *botón*?

Make

```
$ gcc -ansi -Wall -Werror -pedantic -o basics basics.c
```

- ¿Cada vez que quiero compilar?
- ¿De verdad que no hay un *botón*?

make

 Crea el siguiente fichero Makefile con tu editor de texto:

```
basicos: basics.c
    gcc -ansi -Wall -Werror -pedantic -o basics basics.c
```

 Y ejecuta `$ make basics`

 15'

El significado del tipo char

$\llbracket \text{expresión en C} \rrbracket = \text{significado matemático}$

El significado del tipo char

$\llbracket \text{expresión en C} \rrbracket = \text{significado matemático}$

$\llbracket \text{char} \rrbracket = \{-128, -127, \dots, -2, -1, 0, 1, 2, \dots, 127\}$

$\llbracket 97 \rrbracket = 97$

$\llbracket \text{'a'} \rrbracket = 97$

$\llbracket \text{mi_char} \rrbracket = 97$

$\llbracket \text{'a'} == 97 \rrbracket = ?$

Overflows i

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

$$[[c]] = -62$$

💬 ¿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, ...

Cuidado con los *overflows*

Pasa con todos los tipos básicos

Retomemos la coma flotante por un momento



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

(IEEE 754 binary32)

⚠ ¡Sólo fracciones binarias! $\implies$ 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	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pérdida de precisión en la práctica

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

Pérdida de precisión en la práctica

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



Transcribir y ejecutar
(flotante.c)

Pérdida de precisión en la práctica

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

 Transcribir y ejecutar
(flotante.c)

 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:

$$\text{fabs}(f1 - f2) < 0.001$$

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

David Goldberg

unsigned

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

sólo positivos

- Por ejemplo:

$$\llbracket \text{char} \rrbracket = \{-128, -127, \dots, -2, -1, 0, 1, 2, \dots, 127\}$$
$$\llbracket \text{unsigned char} \rrbracket = \{0, 1, 2, \dots, 255\}$$

- Igual para todos los tipos enteros.

¿Todos los caracteres?

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

 15'

long

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

más valores, más precisión

- Se pueden poner dos **longs**:

long long int

- Se puede combinar con **unsigned**

unsigned long int

unsigned long long int

 Aumentar el programa `sizeof.c` con tipos **longs**.

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`
(`locate limits.h`) y explóralo con un editor de texto
(`nano /usr/include/limits.h`)

- Otra parte de la biblioteca: `float.h`

 Busca en tu máquina el *header* `float.h` y explóralo con un editor de texto