

SESIÓN 6: TIPOS BÁSICOS Y MEMORIA

PROGRAMACIÓN PARA SISTEMAS

ÁNGEL HERRANZ

CURSO 2023-2024

OBJETIVOS DE LA SESIÓN

- Familiarizarse con la estructura de un único fichero C
- Tipos básicos: enteros y coma flotante
- Entender el ámbito de variables y funciones
- Entender *dónde* viven las variables y las funciones en ejecución
- Anticiparse a que es importante entender el tamaño de los datos en C
- Estructuras y arrays
- Cada ejemplo introduce funciones de la biblioteca estándar

RECORDATORIO: MINITUTORIAL

- Un fichero C es una colección de
 - ▶ **Includes** con declaraciones de *macros*, variables globales y funciones
 - ▶ Definiciones de **variables globales** (se pueden usar en cualquier función)
 - ▶ Definiciones de **funciones** (las funciones son **globales** y **no se pueden anidar**)
- Los símbolos **globales no se pueden ocultar**
- Las funciones tiene un **tipo de retorno**, **parámetros formales** y **cuerpo** (como los métodos de Java pero en C no hay clases)
- Las declaraciones de variables son **como en Java** y los **tipos enteros** básicos son **char** (1 byte) e **int** (entre 2 y 8 bytes)
- La **llamada a funciones** es con sintaxis clásica: $f(x, y)$
- El **control de flujo** es similar al de Java: **if**, **case**, **for**, **while**, etc.

TIPOS BÁSICOS

- En C los tipos básicos son **enteros y coma flotante**

| Enteros | Coma flotante | Tamaño (en bytes) |
|-------------|---------------|--|
| char | | 1 |
| int | | min. 2, norm. 4 (depende del compilador) |
| | float | 4 (IEEE 754, precisión simple) |
| | double | 8 (IEEE 754, precisión doble) |

- **No** hay booleanos, ni strings (como tales)
- Los enteros son **enteros con signo**

MODIFICADORES: **signed/unsigned**

- Para la **aritmética de enteros: signed/unsigned**
- **Por defecto char y int** son **signed**
- Aritmética **signed**: 125, 126, 127, -128, -127, ...
- Aritmética **unsigned**: módulo 2^n donde n es el tamaño en bits
- Se pueden usar solos:

signed \equiv **signed int**
unsigned \equiv **unsigned int**

MODIFICADORES: **short/long**

| Tipos | Alias | Tamaño (en bytes) |
|----------------------|------------------|--|
| short int | short | min. 2, norm. 2 (depende del compilador) |
| long int | long | min. 4, norm. 8 (depende del compilador) |
| long long int | long long | min. 4, norm. 8 (depende del compilador) |
| long double | | 16 (IEEE 754, precisión doble extendida) |

Nota: para **int** se pueden combinar con modificadores **unsigned**



Escribir un programa en C (`basicos.c`) que declare una variable de cada tipo básico, que imprima en la salida estándar su contenido, el tamaño en bytes ocupado por cada variable en la memoria y su dirección de memoria (compilar con las opciones del examen). Probar con variables globales, locales y parámetros formales de alguna función.

Pistas:

- El operador **sizeof** devuelve el número de bytes que ocupa en memoria **un dato o un tipo** (ej. **sizeof(x)** o **sizeof(long int)**)
- El operador **&** devuelve la dirección de memoria del dato (ej. **&x**)
- La conversión **%p** en el formato de `printf` sirve para mostrar direcciones de memoria (ej. `printf("%p\n", &x)`)
- ¿Qué significará `&main`?



Escribir script de Bash (run.sh) para compilar y ejecutar el programa basicos.c.

```
#!/bin/bash
CFLAGS="-Wall -Werror -ansi -pedantic"
if gcc ${CFLAGS} -o basicos basicos.c; then
    ./basicos
fi
```

Pistas:

- Dar permisos de ejecución: `chmod +x run.sh`
- Ejecutar el script: `./run.sh`

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

Cuidado con los *overflows*

Pasa con todos los tipos básicos

ALGUNAS CARACTERÍSTICAS DE LOS ENTEROS I

- Se usan enteros en las condiciones de **ifs** y bucles:

Igual a 0 ≡ false

Distinto de 0 ≡ true

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

```
f = 1; while (--n) f *= n;
```

ALGUNAS CARACTERÍSTICAS DE LOS ENTEROS II

- ¿Puedes explicar este resultado?

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

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

El char es: 97

El int es: *

- Un caracter es simplemente el símbolo asociado a un número de acuerdo con una tabla: **ASCII**

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 %. |

CUIDADO CON LA COMA FLOTANTE

```
float mi_double = 0.00000001;  
printf("El double es: %f\n", mi_double);
```

El double es: 0.000000

CUIDADO CON LA COMA FLOTANTE

```
float mi_double = 0.00000001;  
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.

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**?



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");
}
```

 desiguales

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");
}
```

 desiguales

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

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

- 0 en código:

$$\text{fabs}(f_1 - f_2) < 0.001$$

What every computer scientist should know about floating-point arithmetic

David Goldberg



Escribir un programa en C (`genmat.c`) que genere matrices de doubles “aleatorios” entre 0 y 1². El programa recibe como argumentos el número de filas y columnas. Imprime la matriz en la salida estándar: una línea con el número de filas (n), otra con el número de columnas (m), y n filas, una por línea, con m doubles separados por espacios³. Para el ejemplo de uso `./genmat 2 3` la salida será:

```
2
3
0.03 0.33 0.69
0.42 0.21 0.25
```

²Usaremos las funciones `rand` y `srand`, con la semilla a 42 y dividiendo por `RAND_MAX`.

³Usaremos el conversor `"%.2f"` en el `printf` para imprimirlos.