ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

C Programming



(some) Data types

R01
Ángel Perles
angel.perles@upv.es

Contents

- Objectives
- Introduction
- (Standard) Integer data types
- Floating-point data types
- Boolean data type
- Enumerated data type
- Collections of elements
  - Arrays
  - Structures
- Const type qualifier

Objectives

- Understand how C evaluates expressions and applies automatic type conversions
- Choose the right data type for each purpose, using the fixed-width types from stdint.h
- Control the type of literal constants (suffixes U, L, F, …)
- Use bool, enum and typedef to write more readable code
- Group data with arrays and structs
- Use const to protect data and to build look-up tables

Introduction

- Please let me know the value of *sweetness* after the execution of this C code

```
double sweetness;

sweetness = 1 / 3;
sweetness = sweetness + 30;
```



Introduction

- And now, explain to me the order in which this C expression is evaluated … and the **automatic conversions** taking place

```
int malapata;

malapata = 2 + 3 * 4.7 / 20;
```



Introduction

- If you understand the mechanics associated with the code in the previous slides, then you can be comfortable with C
  - C is frustrating and error-prone if you don't understand its operational intricacies
  - It is always a good idea to drive its operation and avoid "implicit" matters
  - E.g.: control data types, abuse of parentheses

```
double sweetness;

sweetness = 1.0 / 3.0;
sweetness = sweetness + 30.0;
```

```
int malapata;

malapata = ((2 + 3) * 4.7) / 20;
```

- **NOTE**: malapata's expression result is different from the previous slide, and continues to include non-sense stupidity

Introduction

- Another typical misunderstanding
  - External representation: what you see in the code
  - Internal representation: how this is implemented in the machine

    We assume a 32 bits machine here … hell!

```
int malapata;

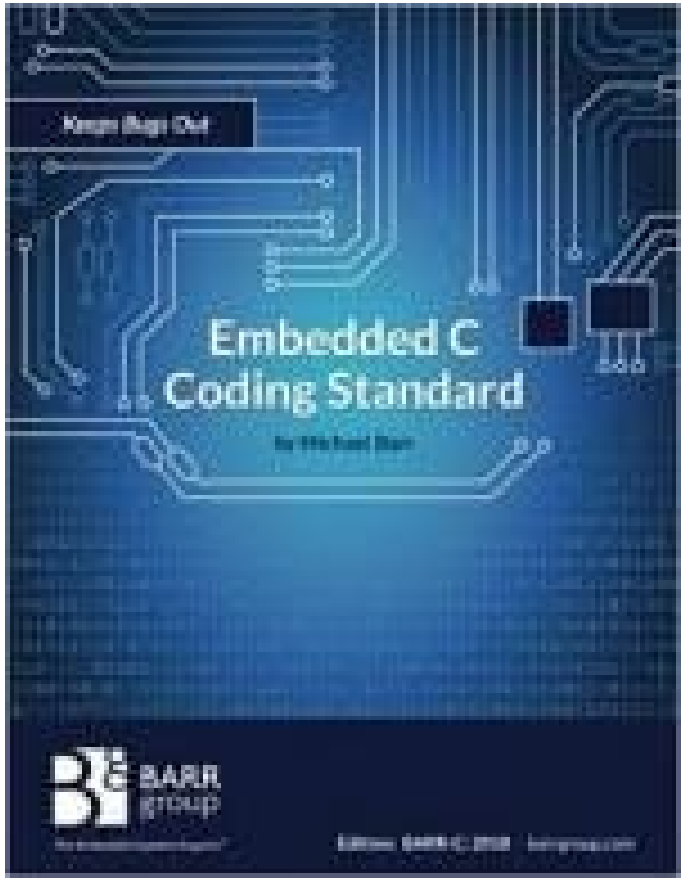
malapata = 65;
malapata = 'A';
malapata = 0x41;
malatapa = 0101;
```

malapata's memory address content

00000000000000000000000001000001

Introduction

- Let's travel the different C data types in a portable and safe way
  - **NOTE**: Most C code available everywhere does not meet this approach



INTEGER TYPES



(Standard) Integers

- *int* is (mostly) gone!
- You **must** use the standard definitions available in *stdint.h* (C99 standard)
- Helps portability, performance and space requirements

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	-2^7 which equals −128	$2^7 - 1$ which is equal to 127
uint8_t	Unsigned	8	1	0	$2^8 - 1$ which equals 255
int16_t	Signed	16	2	-2^{15} which equals −32,768	$2^{15} - 1$ which equals 32,767
uint16_t	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
int32_t	Signed	32	4	-2^{31} which equals −2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
uint32_t	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
int64_t	Signed	64	8	-2^{63} which equals −9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

(Standard) Integers

- **Activity**:
  Create the best-typed array to contain 1000 integers between 3 and 137.
  Develop the code to add all the values and store the resulting value in an appropriate variable
  By now is not necessary to print anything in the screen

  Array syntax in C
  data_type name_of_var [num_elements];
  Remember to include the header file

```
#include <stdint.h>
```

An 1.70 x 1.42 mm St Microelectronics STM32C011F4P6TR microcontroller (ARM Cortex M0+) with 16 KB of flash memory and 6 KB of SRAM





(Standard) Integers

- But, when we write a constant in C …
- e.g.

```
malapata = 3 * alpha;
```

- *3* is an "*int*" by default
- We can use suffixes to control the underlying type

```
1234          // int
1234L         // long, non-portable, is 32 in Windows 11+GCC
1234UL        // unsigned long
2147483647    // int maximum in a 32-bit machine
2147483648    // long long in a 32-bit machine because higher than the maximum
1234ULL       // unsigned long long
```

(Standard) Integers

- Do you know the C operators for integers (and other data types)?
  - Arithmetic

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

(Standard) Integers

- Do you know the C operators for integers (and other data types)?
  - Comparison

Operator	Name	Example	Description
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal
>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

(Standard) Integers

- Do you know the C operators for integers (and other data types)?
  - Logical
  - NOTE: C uses integers to represent boolean concepts

    An integer value = **0 means false**

    An integer value ≠ **0 means true**

Operator	Name	Example	Description
&&	AND	x > 3 && x < 10	Returns 1 if both statements are true
||	OR	x > 3 || x < 4	Returns 1 if one of the statements is true
!	NOT	!(x > 3 && x < 10)	Reverses the result, returns 0 if the result is 1

(Standard) Integers

- Do you know the C operators for integers (and other data types)?
  - Bitwise operators (operations at the bit level)
  - A must for embedded programming. Not necessary for the rest

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if at least one of the bits is 1
^	XOR	Sets each bit to 1 if only one of the bits is 1
~	NOT	Inverts all the bits
<<	Left Shift	Shifts bits to the left (multiplies by powers of 2)
>>	Right Shift	Shifts bits to the right (divides by powers of 2)

(Standard) Integers

- Important notes for Java users and other happy ones
- Java does not have unsigned integers
  - E.g. this dangerous expresion is false (executes the *else* part)

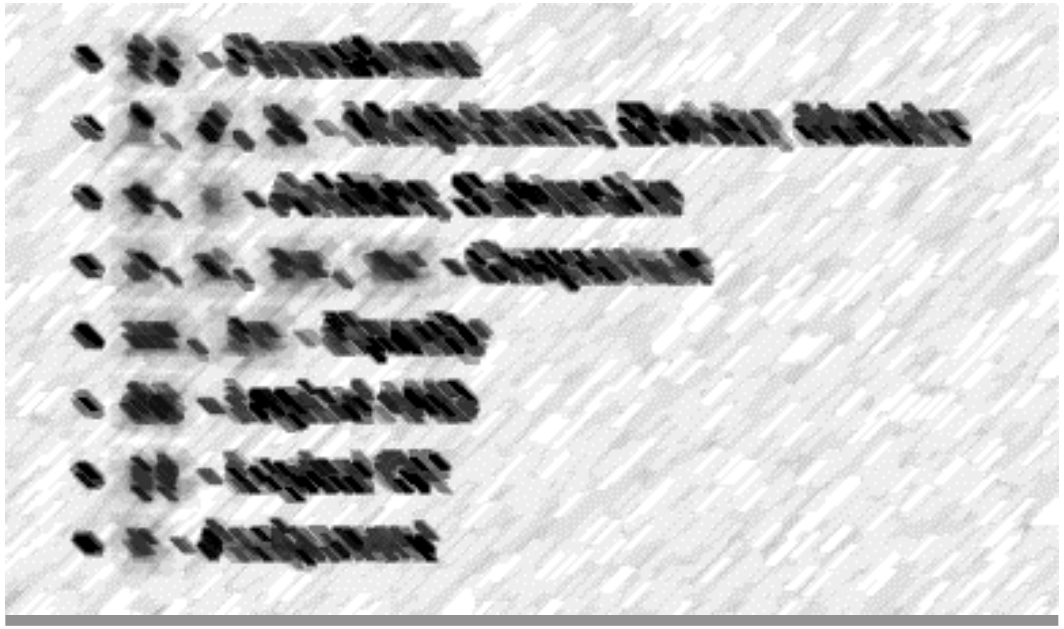
```
if (-1 < 1U)
```

- This is a typical dangerous error: it is not a comparison; it is an assignment and then check *hell*

```
if (hell = 3)
```

(Standard) Integers

- Each operator in an expression has a precedence (priority)
  - But always use parentheses to avoid remembering the precedence order (Barr-C rule)
  - Check the precedence of some operators in the following table ;-)



FLOATING-POINT DATA TYPES

Floating-point data types

- The most typical floating-point data types in C are
  - *float* for single precisión
  - *double* for double precision

- And the internal representation will be compiler/machine dependent
  - Perhaps it is based on IEEE754 simple and double precisión format
  - Assuming that this is true in your target platform, we have

float size -> 32 bits = 4 bytes
double size -> 64 bits = 8 bytes

- NOTE: Some compilers for microcontrollers use *float* internally when you use *double*. Take care

Floating-point data types

- Deciding between integer/float/double is crucial for a successful implementation
  - Size constraints
  - Execution time constraints
  - E.g., microcontroller-based systems, artificial-intelligence models
- Let's assume that we want to use float/double
  - Then **you must be aware of the "F" suffix**
  - E.g. Cortex-M4F microcontrollers only have single-precision FPU, so doubles are computed using software

BAD

```
float a;
a = 3.0;
a = a + 2.7;
```

OK

```
float a;
a = 3.0F;
a = a + 2.7F;
```

Floating-point data types

- **Activity (revisited)**:
  Create the best-typed array to contain 1000 temperaturas between -3.0 and 15.0 degrees Celsius.
- Use IA to generate the code to randomly fullfill the array with randon values between the admited ranges.
- Develop the code to count the values below 0 and greater than or equal than 0
- Add all the values and store the resulting value in an appropriate variable

Array syntax in C

data_type name_of_var [num_elements];



BOOLEAN DATA TYPE

Boolean data type

- I revisit the idea that C uses integers to represent boolean concepts where
  - An integer value = 0 means false
  - An integer value <> 0 means true (so, 3, -5, 1997 mean TRUE)
- C99 introduced the **bool** data type to enhance abstraction
  - Requires adding the following header file

    #include <stdbool.h>

  - And admits the values **true** and **false**

```c
#include <stdbool.h>

bool is_working_day = true;

if (is_working_day)
{
    // Code to execute if it is a working day
}
else
{
    // Code to execute if it is not a working day
}
```

ENUMERATED DATA TYPE

Enumerated data type

- Among others, **enumerated data type** bridges the dilemma between the ease of representing integers on a computer and the human ability to mentally represent "mnemonic" words
- Internally, they are integer types
- Let’s discuss how to represent the state of a open/close valve (digital valve)



Enumerated data type

- Let's discuss how to represent the state of an open/close valve (digital valve) …
- A possible solution using "**enumerated**" and a new data type definition (**typedef**)
  - We are defining a new data type named *valve_state_t*
  - The *_t* is not accidental, it is a human approach to say "this is a type"

```
typedef enum
{
    VALVE_OPEN,
    VALVE_CLOSED
} valve_state_t;
```

```
valve_state_t irrigation_valve_state;

irrigation_valve_state = VALVE_OPEN;
...
if (irrigation_valve_state == VALVE_CLOSED)
{
    printf("La valvula esta cerrada.\n");
}
```

Enumerated data type

- enums provide human readability to the code
- Activity:
  - Create an enumerated type to represent the days of the week
  - Create a variable of that type and assign it the value "Monday“
  - NOTE: The order of the list of enumerations is important (I’ll explain it later)



COLLECTIONS OF ELEMENTS

Collections of elements: arrays

- **arrays** in C are fixed-size collections of elements of the same type
- It is defined using the following (generalized) syntax

data_type name_of_the_variable [num_of_elements_1][num_of_elements_2]…;

- Where *num_of_elements_X* defines the number of elements for a given dimension
- E.g., a variable to store 1000 doubles

```
double values [1000];

// ...
values[0]   = 3.14;   // first element
values[999] = 4.5;    // last element
```

- **NOTES**: the first element has index 0; you cannot use = for copy; accessing out of limits works!!!!; no .length available like in Java

Collections of elements: structs

- **structs** in C are a way of grouping variables of the same or different types
- They use the syntax

```
struct structure_name
{
    data_type_1 var_name_1;
    ...
    data_type_N var_name_N;
};
```

- But I suggest this approach: combine it with the definition of a new type

```
typedef struct
{
    data_type_1 var_name_1;
    ...
    data_type_N var_name_N;
} new_type_t;
```

Collections of elements: structs

- E.g., representing complex numbers

```c
typedef struct
{
    double real;
    double imaginary;
} complex_t;

complex_t point;        // create a variable of type complex_t
complex_t tmp;

point.real = 10.0;      // accessing field "real"
point.imaginary = 3.4;  // accessing field "imaginary"

tmp = point;            // copies the values, NOT the reference
```

Collections of elements: structs

- E.g., representing complex numbers (cont.)
  - You can also create an array based on this new type

```
#define NUM_ITEMS 50

complex_t items [NUM_ITEMS];

for (int index=0;index < NUM_ITEMS;index++)
{
    items[index].real = 3.0;
    items[index].imaginary = (double)index + 1.5;
}
```

  - Note the **typecast** *(double)index* is not mandatory, but explicit = good

CONST PREFIX

Const type qualifier

- Data type creation can include type qualifiers that modify their behaviour
- One of them is *const*, which marks “something” as read-only (can’t be modified)
- Two interesting uses of this prefix are:
  - Avoid changes to objects, providing robustness to the code
  - Usually store the objects in the code segment, saving data space (perfect for microcontrollers)
- E.g.

```
const double acceleration_earth = 9.81;
const double acceleration_moon  = 1.62;
```

- This has some advantages compared with preprocessor directives (data type,…)

```
#define ACCELERATION_EARTH 9.81
#define ACCELERATION_MOON  1.62
```

Const prefix

- More interesting *const* examples
  - Note here the creation of an array of predefined values. The size is calculated by the compiler based on the provided data
  - Examples below are named “look-up” tables

```
const uint16_t heat_table[]={327,2935,9512,536,9254,554,
                             9769,2,0,2314,57477,345,1900,1794};
```

```
const float sin_table [] = {0.0F, 0.0017F, 0.0034F, ...};
```

- You can use *sizeof(table) / sizeof(table[0])* to compute the number of elements of a given array

Const prefix

- More interesting *const* examples (cont.)
  - "Look-up" table for a 3x8 digital decoder

decimal	C	B	A	b7	b6	b5	b4	b3	b2	b1	b0	hexadecimal
0	0	0	0	0	0	0	0	0	0	0	1	01h
1	0	0	1	0	0	0	0	0	0	1	0	02h
2	0	1	0	0	0	0	0	0	1	0	0	04h
3	0	1	1	0	0	0	0	1	0	0	0	08h
4	1	0	0	0	0	0	1	0	0	0	0	10h
5	1	0	1	0	0	1	0	0	0	0	0	20h
6	1	1	0	0	1	0	0	0	0	0	0	40h
7	1	1	1	1	0	0	0	0	0	0	0	80h

```
const uint8_t deco3x8_table [] = {0x01, 0x02, 0x04,
                                  0x08, 0x10, 0x20, 0x40, 0x80};
```

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA