

Embedded Real Time Control (ERTC)

C programming language

Alberto Morato, Mauro Tubiana

What is C?

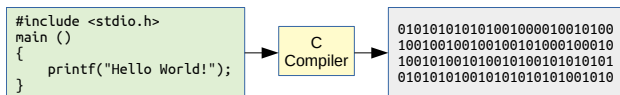
- C is programming language that combines the power of **Assembly** language with the readability and maintainability of **High level programming languages**.
- (Probably) it is the most effective and efficient language when you have to interact with hardware from a software prospective (e.g. in embedded systems).

History of C

- C was created in the 1969 at Bell Labs by Dennis Ritchie
- Dennis Ritchie wrote, together with Brian Kernighan, the C Bible: The C Programming Language
- Despite it is an "old" language, it is in continuous evolution
 - It is a standardized language
 - Last version ISO/IEC 9899:2018 (C17)

What is C?

- Is a high level programming language
- Is a simple language
- Is very close to machine language
- Is a compiled programming language



Why C?

- It is a fast and low overhead language
- No garbage collection
- No runtime checks (e.g. division by zero, out-of-bounds, etc.)
- It allows low level control of the computer, especially memory
- Light executable

Why not C?

- Can be unsafe
 - Actually, people that use C in the "wrong" way make it unsafe
- Safety and resources management is a burden for the programmer
- Abstractions are more difficult with C than with other programming languages

Structure of a C program

```
#include <stdio.h>
#define PI 3.141592

const float e = 2.71828;

int counter = 0;

int main()
{
    printf("Hello_world!\n");
}
```

A C program consists of:

- Inclusion of libraries
- Definition of constants
- Definition of variables
- Definition of functions

Structures of a C program

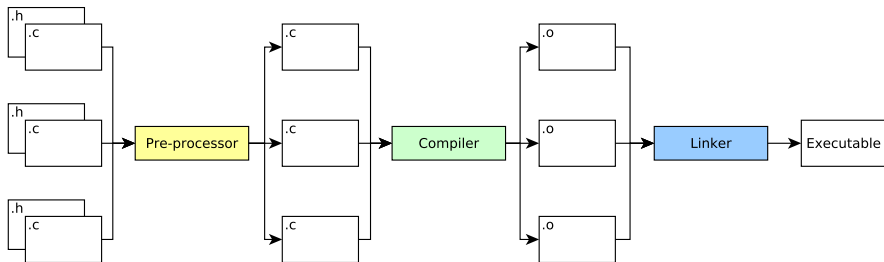
- Every C program consists of one or more functions and variables
- `main` is a special function
 - It is the entry point of the program
- Functions contains statements, which specify the operations to be done
- Variables store values used during the computation
- Statements ends with a semicolon
- "Identifiers" are the names you supply for variables, types, functions, and labels in your program

Structure of a C project

A C project may consists of multiple .c and .h files.

- .h files, called header files, contain the declaration of functions and variables that you want to share among one or multiple .c files.
- .c files contain the actual implementation of the functions.

Compile process



Compile process

- A compiler produces a program that can run into a specific processor, for example, a program that you compile for your computer cannot run, in general, on your phone
- We are going to mostly use a special kind of compiler, called cross-compiler, which is compiler able to create executable files for platform or architecture other than the one where the compiler is running

Write a C program

What you need is:

- A text editor (e.g. Notepad)
- A compiler (e.g. gcc)
- Patience and goodluck



- There are more sophisticated environments called IDEs (Integrated Development Environments) that provide advanced functionality (e.g. debugging, memory inspection and manipulation)
- In lab we will use STM32CubeIDE

Comments

- Single line

```
// this is a comment
```

- Multi line

```
/* This  
is a  
multi line comment  
*/
```

- A comment is ignored by the compiler

Variables

Variables store data that are needed during the computation.

```
variable_type variable_name = value;
```

A C variable has:

- A data type
 - specify what type of data the variable can contain;
 - basic data types in C are: char, short, int, long, float, double (all with sign)
 - you can declare variables without sign: unsigned char, unsigned int ...
 - different types have different sizes
- A name
 - cannot use keywords as variable names
 - name cannot start with a number
- can have a value assigned to it.
- an address: every variable has a location in memory where it lives;

Variables

- `float x;`
`char exit_character;`
- You can declare multiple variable of the same type like this:

```
float x, y, z;
```

- `float x = 10.0;`
- `float x = 10.0, y = 20.0, z = 30.5454;`
- `int a;`
`int b = 10;`
`a = b;`
- `char exit_character;`
`exit_character = 'q';`



```
x = 10;
```

Variables

Every **type** is associated with a specific size, i.e. the number of bytes (or bit) that it occupies in memory. $|T| = 2^n$

Type	Size (bit)	Min	Max
char	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807
unsigned char	8	0	255
unsigned short	16	0	65535
unsigned int	32	0	4294967295
unsigned long	64	0	18446744073709551615
float	32	1.17549e-38	3.40282e+38
double	64	2.22507e-308	1.79769e+308


- Exceeding the limits leads to overflow.

 Same type in different architectures may have different sizes.

Variables

The standard library `stdint.h` provides a set of types that are guaranteed to be the same size on all platforms.

```
int8_t  
int16_t  
int32_t  
uint8_t  
uint16_t  
uint32_t
```

 The use of `stdint.h` is highly encouraged.

Variables

The size of a variable can be obtained using the function

```
sizeof(...);
```

for example

```
sizeof(int); // returns 4
```

or

```
int var;  
sizeof(var); // returns 4
```

Operators and expressions

- **operators** specify what is to be done with variables;
- **expressions** can combine variables and other elements, like constants, to produce new values;

Operators

- Arithmetic Operators

- `+, -, *, / and %`

- Relational Operators

- `>, <, >=, <=`

- Equality Operators

- `==, !=`

- Logical Operators

- `||, &&, !`

Operators and expressions

Increment and decrement operator

- `x++;`

- `x--;`

Operator precedence

- see the reference to the C programming language to know the order in which operator are taken into account
- precedence rules can be altered using parenthesis ();

Operators and expressions

Assignment and expressions

```
x = x + 2;
```

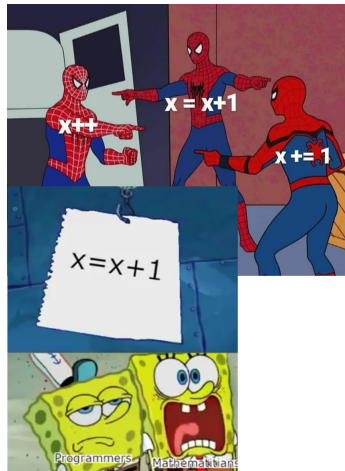
and

```
x += 2;
```

are equivalent.

Same rule applies to binary operators;

⚠ the expression above does not have the meaning of a mathematical equation



Statements and blocks

- An expression such as `x = 0` or `i++` or `printf (. . .)` becomes a statement when it is followed by a semicolon, as in

```
x = 0;  
i++;  
printf(...);
```

- Braces `{` and `}` are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement.
- There is no semicolon after the right brace that ends a block.
- In general: The bodies of C functions are made up of statements. These can be either simple statements that do not contain other statements, or compound statements that have other statements inside them

Type conversion

- C allow to convert a variable of one type to a different type by mean of the cast operator ()

```
int c = 10;  
float f = (float)c; // f contains 10.0
```

- Type conversion can be implicit, which means that is automatically applied by the compiler

? What happens if a float variable is casted to int?

example: `basic_conversion.c`

Type conversion

```
int number1 = 2;  
int number2 = 3;  
float result;  
result = number1 / number2;
```

The value of result is 0;

Type conversion

```
int number1 = 2;  
int number2 = 3;  
float result;  
result = (float)number1 / (float)number2;
```

The value of result is 0.66.....

Same result also casting only one of the two variables

example: `basic_conversion_division.c`

if / else

- Used to create conditional branches in a program.

```
if (expression)
    statement_1
else
    statement_2
```

- else is optional
- If expression is true (i.e. has a non-zero value), statement_1 is executed. Otherwise, if it is false (expression is zero) it is present the else part, statement_2 is executed.

If / else

```
int x = 5;  
int y = 10;  
int z = 20;  
  
if (x < 3)  
    y = 0;  
    z = 0;
```

? What is the value of y and z?

example: pitfall_if.c

If / else

There are shortcuts:

```
if (n != 0)
{
    ...
}
```

is equivalent to

```
if (n)
{
    ...
}
```

If / else

Concatenation of multiple if / else

```
if (expression)
{
    statements
}
else if (expression)
{
    statements
}
else if (expression)
{
    statements
}
else
{
    // optional
}
```

Switch

- The switch statement is a multi-way decision that tests whether an expression matches one of a number of integer values, and branches accordingly.

```
switch (expression) {  
    case value1:  
        statements  
        break;  
  
    case value2:  
        statements  
        break;  
  
    default: // default is optional  
        statements  
        break;  
}
```

Switch

- somehow equivalent to the if / else if / else expression seen before but more elegant;
- the statement inside default is executed if none of the above cases match the condition;
- Example: `switch-example.c`

While loop

- a while loop takes the general form of

```
while (expression)
{
    statements
}
```

- a do / while form exists (rarely used in practice)

```
do
{
    statements
}
while (expression);
```

- the expression is evaluated. If it is non-zero, statement is executed and expression is re-evaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.

Example While loop

```
int i = 0;
while (i < 10)
{
    // do something here
    i++;
}
```

- Here we have done something 10 times

For loop

a for loop takes the general form of

```
for (expr1; expr2; expr3)
{
    statements
}
```

and is basically equivalent to the while loop below

```
expr1;
while (expr2)
{
    statements
    expr3;
}
```

Example For loop

```
int i;  
for(i = 0; i < 10; i++)  
{  
    // do something here  
    /* this comment and this code works in every version of C */  
}
```

```
for(int i = 0; i < 10; i++)  
{  
    // do something here  
    // this comment and this works only in C99 and later versions  
}
```

- Here we have done something 10 times

Infinite loop

- ```
while (true)
{
 ...
}
```

- ```
while (1)
{
    ...
}
```

- ```
for (; ;)
{
 ...
}
```

- an infinite loop can be exited using return or break

# Loops

- `break` and `return` exits the loop;
- `continue` does not execute the successive instructions inside the loop and start a new iteration instead;
- loops can be nested;
- Example: `continue-example.c`
- Example: `nested-loop-example.c`

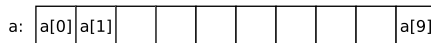
# Goto

- goto allow for local jump using labels;
- considered evil;
- goto is easily abusable and can lead to unstructured and difficult to maintain source code (so called spaghetti code);
- may be useful in some cases as in error management;
- Example: goto-example.c



# Arrays

- Arrays are a collection of continuous and homogeneous elements



- Array declaration:

```
array_type array_name[number_of_elements]
int foo[10];
```

- number\_of\_elements is optional but then you have to initialize the array

```
int bar[] = {0,0,0,0,0,0,1,15,98,65};
```



# Arrays

## Accessing elements

```
int foo[10];
foo[3] = 5;

int temp;
temp = foo[1];
```

⚠ The access range is `[0, number_of_elements-1]`

- to get the number of byte occupied by the array

```
sizeof(array_name)
sizeof(foo) // gives 10*4=40 in many cases
```

- to get the number of elements in the array

```
sizeof(array_name)/sizeof(array_type)
sizeof(foo)/sizeof(int) // gives 10
```

- Example array-size.c

# Array

## Array initialization

```
int bar[10] = {0,0,0,0,0,0,1,15,98,65};
int bar[] = {0,0,0,0,0,0,1,15,98,65};
```

- the two initializations are equivalent
  - in the second case the compiler automatically decide the array

⚠ In C you can read and write outside the array boundaries

```
temp = bar[-1] // is not like python
bar[120] = 12;
```

⚠ especially if you **write** outside the boundaries you are gonna have bad times (memory corruption)

# Arrays

Array of char a.k.a. string

- Strings in C are arrays of char with a implicit special character added as last element (0x00 or "\0")

```
char my_string[5] = { 'H', 'e', 'l', 'l', 'o' };
char my_string[] = "Hello";
```

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| H | e | l | l | o | \0 |
|---|---|---|---|---|----|

- <string.h> contains some functionalities to manipulate strings in C
- The char type can be safely used to represent 7-bit character encodings, such as ASCII

# Arrays

## Multi-dimensional arrays

- C provides rectangular multi-dimensional arrays

- ```
int foo[3][4] = {  
    {0,1,2,3},  
    {4,5,6,7},  
    {8,9,10,11}  
};
```

- ```
int bar[2][2][2] = {
 {
 {0,1},
 {2,3}
 },
 {
 {4,5},
 {6,7}
 }
};
```

# Functions

- Functions are a way to encapsulate some computation which can be used without worrying about its implementation;
- Are one of the main ways to enforce code reuse and program structure in C;
- Functions allow to break large tasks into smaller ones;

# Functions

## Function declaration

Instruct the compiler about a function's name, return type, and parameters

```
return_type function_name(argument_type argument_name, ...)
```

Usually it is placed in the .h file

## Function definition

It is the actual implementation of the function

```
return_type function_name(argument_type argument_name, ...)
{
 declarations and statements
}
```

Usually it is placed in the .c file

# Functions

- A program is just a set of variables definitions and functions.
- Communication between the functions is performed by arguments and return value.
- Functions definitions don't require a specific order
- The program can be split into multiple files (.c and .h)
  - A function definition cannot be split
- There is always a function caller and a callee
  - To be called a function needs to be defined

# Functions

## Example of function declaration

```
int f(void);
int *fip();
void g(int i, int j);
void h(int, int);
int square(int);
```



# Functions

## Example of function definition

```
int square(int num)
{
 int result;
 result = num * num;

 return result;
}
```

# Functions

## Calling a function

Considering the square function on the previous slide, to call the square function:

```
int main(void)
{
 int number = 5;
 int result = square(5);
 return 0;
}
```

calling a function means to run the instructions that the functions contains

Example: `basic.c`

# Functions

## Return statement

The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return:

```
return expression;
```

The expression will be converted to the return type of the function if necessary.

# Functions

## Argument passing

- In C, all function arguments are passed “by value.” This means that the called function is given the values of its arguments in temporary variables rather than the originals.
- Function cannot directly alter a variable in the calling function; it can only alter its private, temporary copy.
- Example: `argument-passing-example.c`

# External variables

- “internal” variables are those declared inside a function and the function’s arguments
- External variables are defined outside of any function, and are thus potentially available to many functions.
- external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.
- If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists BUT can lead to programs with too many data connections between functions

# Scope

- Variables and functions identifiers (among others) have a scope that delimits how those elements can be accessed
- In general the scope of a variable or a function identifier is determined by where it is declared
- C has four types of scope:
  - file;
  - block;
  - function prototype;
  - function;

# Scope

## File Scope

- Obtained when the declaration is outside of any block or parameter list (the list of arguments in a function declaration)
- The scope of the element taken into account is the entire file in which it appears
- Example: `file-scope-example.c`

## Block scope

- If the declaration appears inside a block or within the list of parameters, it has block scope
- The identifier it declares is accessible only within the block
- Example: `block-scope-example.c`

# Scope

## Function prototype scope

If the declaration appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has function prototype scope which terminates at the end of the function declaration.

## Function scope

- is the area between the opening of a function definition and its closing
- is applicable to labels only (see goto)



# Scope

- Scopes can be nested, with inner and outer scopes. For example, you can have a block scope inside another block scope, and every block scope is defined within a file scope. The inner scope has access to the outer scope, but not vice versa. As the name implies, any inner scope must be completely contained within the outer scopes that encompass it
- If you declare the same identifier in both the inner scope and an outer scope, the identifier declared in the outer scope is hidden by the identifier within the inner scope, which takes precedence

## Example of scope

```
// file scope of j begins
int j;
// block scope of i begins
void f(int i)
{
 // block scope of j begins; hides file-scope j
 int j = 1;
 // i refers to the function parameter
 i++;
 for (int i = 0; i < 2; i++)
 {
 // block scope of loop-local i begins
 int j = 2; // block scope of the inner j begin; hides o
 printf("%d\n", j); // inner j is in scope, prints 2
 }
 // block scope of the inner i and j ends
 printf("%d\n", j); // the outer j is in scope, prints 1
}
// the block scope of i and j ends
void g(int i): // i has function prototype scope; hides file-s
```

# Lifetime

- The lifetime of a variable or object is the time period in which the variable/object has valid memory.
- Lifetime is also called “allocation method” or “storage duration.”
- The main storage durations in C are:
  - automatic
  - static

# Lifetime

## Automatic storage duration

- An automatic variable has a lifetime that begins when program execution enters the function or statement block or compound and ends when execution leaves the block
- Function parameters have automatic storage duration

## Static storage duration

- The object's lifetime is the entire duration of the program's execution.
- Example: `static-example.c`

# Scope and lifetime

Scope and lifetime are different concepts. Scope applies to identifiers, whereas lifetime applies to objects. The scope of an identifier is the code region where the object denoted by the identifier can be accessed by its name. The lifetime of an object is the time period for which the object exists