

# Computer Architecture (Lab Class)

## *Modules and Makefiles*

Luís Nogueira

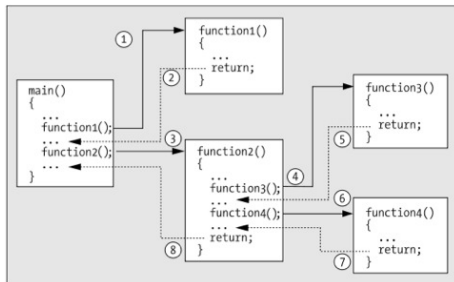
Departamento de Engenharia Informática  
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2024/2025

# Structure of a C program

- A C program is usually divided into functions
- A function is a block of code that performs a specific task. Functions can call other functions
- Every C program must have a `main` function which is the starting point of the program execution



# Three aspects of a C function

- Function declaration (or prototype)
  - Informs the compiler about the function's name, parameters and return value's data type
- Function definition
  - Contains all the statements to be executed
- Function call
  - An expression that passes control and arguments (if any) to a function

## Important note

- Functions should either be declared or defined before being called

# Function declaration (I)

- Function name
  - Defined by the programmer. It can be anything, however it is advised to have a meaningful name
- Return type
  - The data type returned from calling the function (void if no return)
- Function parameters
  - The inputs needed for the function to execute. There can be multiple parameters separated by a comma
  - It is only mandatory to specify their types, not their names
- Example:

```
int average(int x, int y);  
int average(int, int);
```

## Function declaration (II)

main.c

```
1  #include <stdio.h>
2
3  int average(int , int );
4
5  int main(){
6      int v[] = { 1, 2 };
7      int r = 0;
8
9      r = average(v[0], v[1]);
10
11     printf("average =%d\n", r);
12     return 0;
13 }
14
15 int average(int n1, int n2) {
16     return (n1+n2)/2;
17 }
```

- This code does not compile if we omit line 3. Why?

- It is common practice to have a *header* file (*.h*) to each code file (*.c*) with the prototypes of the functions defined in the program
- We can say that each code file *.c* and its corresponding header file *.h* compose a module of the program. Several times the term “compilation unit” is used
- A program can have several modules

main.c

```
#include <stdio.h>
#include "average.h"

int main(){
    int v[] = {1,2};
    int r = 0;

    r = average(v[0], v[1]);
    printf("average = %d\n", r);
    return 0;
}
```

average.h

```
int average(int n1, int n2);
```

average.c

```
int average(int n1, int n2){
    return (n1+n2)/2;
}
```

# Compile a program with several modules

- Modules must be compiled first, followed by the main program

- Command line sequence:

```
/* Build object for the average module */  
gcc -g -Wall -Wextra -fanalyzer  
    -c average.c -o average.o
```

```
/* Build object for the main module */  
gcc -g -Wall -Wextra -fanalyzer  
    -c main.c -o main.o
```

```
/* Build final program */  
gcc main.o average.o -o prog_avg
```

- Execute the final program:  
./prog\_avg

average.h

```
int average(int n1, int n2);
```

average.c

```
int average(int n1, int n2){  
    return (n1+n2)/2;  
}
```

main.c

```
#include <stdio.h>  
#include "average.h"  
  
int main(){  
    int v[] = {1,2};  
    int r = 0;  
  
    r = average(v[0], v[1]);  
    printf("average =%d\n", r);  
    return 0;  
}
```

# Variable scope

- A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration
- If a variable is declared inside a function it can only be used inside that function. It is called a *local variable*
- If it is declared inside a block ("`{ ... }`"), it exists only inside that block

```
{
    int a = 0; /* Declaration of variable "a" */
    /* Refer to "a" here is not an error */
    /* Refer to "b" here is an error! */

    {
        int b = 10; /* Declaration of variable "b" */
        /* One can refer to "a" and "b" here */
    } /* Variable "b" "dies" */

    /* Refer to "b" here is an error! */
    /* Refer to "a" here is not an error */
} /* Variable "a" "dies" */
```



- In C, a program that consists of multiple source code files is compiled one at a time
- Until the compilation process, a variable can be described by its scope
- It is only when the linking process starts, that the linkage property comes into play
- Thus, scope is a property handled by the compiler, whereas linkage is a property handled by the linker
- Should a variable be available for another file to use? Should a variable be used only in the file declared? Both questions are decided by the linker

- A variable or function implementing internal linkage is not accessible outside the compilation unit it is declared in
- It is implemented by the keyword `static`
- The following example gives a linking error. Why?

main1.c

```
#include <stdio.h>
#include "average1.h"

int main(){
    int v[] = { 1, 2 };
    int r=0;

    r = average(v[0], v[1]);
    printf("average = %d\n", r);
    return 0;
}
```

average1.h

```
int average(int n1, int n2);
```

average1.c

```
static int average(int n1, int n2){
    return (n1+n2)/2;
}
```

- Externally linked identifiers are shared between translation units and are considered to be located at the outermost level of the program
- In practice, this means that we must define an identifier in a place which is visible to all, such that it has only one visible definition
- It is the default linkage for globally scoped variables and functions
- The keyword *extern* implements external linkage
- When we use the keyword *extern*, we tell the linker to look for the definition elsewhere

- In the following example, the header file *average2.h*, that is included in *main2.c*, declares that an extern variable with name *result* exists, defined and initialised in *average2.c*

main2.c

```
#include <stdio.h>
#include "average2.h"

int main(){
    int v[] = { 1, 2 };

    average(v[0], v[1]);
    printf("average = %d\n", result);
    return 0;
}
```

average2.h

```
void average(int n1, int n2);
extern int result;
```

average2.c

```
int result = 0;

void average(int n1, int n2) {
    result = (n1+n2)/2;
}
```

# Multiple inclusion problem

defs.h

```
typedef struct {
    int x;
    int y;
} point;

typedef struct car{
    int id;
    point start, end;
    int distance;
};
```

main.c

```
#include <stdio.h>
#include "distance.h"
#include "defs.h"

int main(){

    struct point a,b;
    struct car c;
    ...
    fill_car(c);
    return 0;
}
```

distance.h

```
#include "defs.h"

int distance(point a, point b);
void fill_car(car c, point a, point b);
```

distance.c

```
int distance(point a, point b){
    ...
}

void fill_car(car c, point a, point b){
    ...
}
```

## The problem of multiple inclusions

- Note that defs.h will be included twice in main.c

# Solving the multiple inclusion problem

- The header should be wrapped in *header guards* to prevent multiple inclusion
- Header guards are implemented through the use of preprocessor directives: `#ifndef`, `#define` and `#endif`
- If the symbol is defined then this section of code has been seen before and should not be processed again. If the symbol has not been created, then the code it is associated with has not been seen
- The name of the symbol must be unique. A common approach is to use the name of the header file, converting the `.h` suffix to a `_H`

defs.h

```
#ifndef DEFS_H
#define DEFS_H

typedef struct {
    int x;
    int y;
} point;

typedef struct car{
    int id;
    point start, end;
    int distance;
};

#endif
```

- `make` is a build automation tool that automatically builds executable programs and libraries from source code by reading files called `Makefiles` which specify how to derive the target program
- The `Makefile` specifies a set of rules to be accomplished and their possible dependencies
- For each rule, if any of the prerequisites has a more recent modification time than the target, the command lines are run
- This means that `make` only recompiles outdated files, shortening the needed compilation time (particularly noticeable in large projects)

- The content of the Makefile must follow a specific syntax:

## Makefile

```
default_rule: dependency1 dependency2 . . .
    <tab>command1
    <tab>command2
<empty line to break consecutive rules>
dependency1: dependency1_1
    <tab>command1
    ...
<empty line to break consecutive rules>
...
```

- A rule can have zero or more dependencies
- Each command line must start by the <tab> character
- An empty line must be used to set apart the last command of a rule from the declaration of the following rule



## Makefile

```
# A simple Makefile
prog_avg: average.o main.o
    gcc average.o main.o -o prog_avg

average.o: average.c average.h
    gcc -g -Wall -Wextra -fanalyzer -c average.c -o average.o

main.o: main.c average.h
    gcc -g -Wall -Wextra -fanalyzer -c main.c -o main.o

clean:
    rm -f *.o prog_avg

run: prog_avg
    ./prog_avg
```

- `prog_avg` is the default rule of this Makefile
- `average.o` and `main.o` are the dependencies to achieve that goal (and also rules by themselves)
- `gcc average.o main.o -o prog_avg` is the command that will be executed in the shell to reach the program goal
- If the dependencies `average.o` and `main.o` haven't yet been executed or are out of date, their respective commands will be executed prior to the commands in to build `prog_avg`
- The rules `clean` and `run` are not prerequisites of any other rule. The only purpose of these rules is to simplify the execution of the program and the removal of the executable file and all the object files

# A more advanced Makefile with variables

## Makefile

```
# Variables
INCLUDES = average.h
SOURCES = average.c main.c
OBJFILES = average.o main.o
EXEC = prog_avg

# Suffix rules
.SUFFIXES : .c .o

# How to build an object .o from a code file .c ; $< -- file name
.c.o:
    gcc -g -Wall -Wextra -fanalyzer -c $<

# How to build the executable from object files
${EXEC}: ${OBJFILES}
    gcc ${OBJFILES} -o ${EXEC}

# Object file dependencies on source and header files
average.o: average.c average.h
main.o: main.c average.h

# Run the program
run: ${EXEC}
    ./${EXEC}

# Clean up generated files
clean:
    rm -f ${OBJFILES} ${EXEC}
```

- To execute what is specified in the Makefile it is a matter of simply invoking `make`  
`make [-f <makefile>] [<rule>]`
- If the `-f` parameter is not used to specify which rule file to use, `make` will search in the current directory for a file named “makefile” or “Makefile”
- By default, the goal is the first rule in the Makefile
- However, any rule in the Makefile may be specified as a goal
- It is recommended to have a different directory to each practical exercise with the needed modules and the respective Makefile

## Makefile

```
# A simple Makefile
prog_avg: average.o main.o
    gcc average.o main.o -o prog_avg

average.o: average.c average.h
    gcc -g -Wall -Wextra -fanalyzer -c average.c -o average.o

main.o: main.c average.h
    gcc -g -Wall -Wextra -fanalyzer -c main.c -o main.o

clean:
    rm -f *.o prog_avg

run: prog_avg
    ./prog_avg
```

- Which commands will be executed in each of these make invocations?
  - make
  - make run
  - make clean