

Introducing C Coding

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q1

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

The goal of this session is mainly introducing basic background to develop, compile and execute codes in C/C++. Actually, in this course we are going to focus on **C coding**. To give you additional support, we attach a link to the [C/C++ language reference web site](#).

The first part of this session presents key background aspects to develop C based programs. To do this, we explain how to implement a simple application. The second part shows the procedure to generate an executable from a C source code. This step is a.k.a. compilation. Take into account this procedure is similar to any compiled language, but it is a totally different approach compared to interpreted languages, such as Python.

Secondly, the lab session addresses the variety of data types discussed during the first topic of Theory lectures. In particular, you are going to play with scalar, floating point, and compound data types. The main objective is to get experience on the differences among them, about their relation with the Programming Language, as well as the software and hardware execution environment.

C Background

We encourage to use a text editor that makes you feel comfortable to work with. There is no specific development framework we recommend to use in Linux. Nevertheless, we suggest to edit code files on frameworks that colour keywords based on the programming language that you are using to code. The editor identifies the language based on the extension of the source code file. For example, “test.c” implies it is a C based source code, whilst “test.cc” or “test.cpp” is based on C++ (inside or outside a Linux/Unix OS, respectively).

As the first steps to get experience on coding, we will start by developing our first “Hello World!” program. Secondly, we will continue understanding the translation step from source code files to executable binary files. Finally, we will conclude this introductory part by automating parts of the compilation phase.

The first C Source Code

Let's create a first source code, called “hello.c”. This program will show a “Hello World!” -like message to the screen. Firstly, develop the main function, called “main”. This is the first function to be executed when the program is launched and should have the following header:

```
int main (int argc, char **argv)
{
    // body;
}
```

By default, the main function returns an “integer” (a.k.a. “int”). The function has two input parameters: “argument counter” (a.k.a. “argc”) and “argument values” (a.k.a. “argv”). The former is an integer that shows how many arguments has the process (the instance of this program launched from the command line) and the latter is a matrix of characters (a.k.a. “char **”). That is, multiple rows with an array of chars each (you may find more information about this type of data in the slides 44-48 of the

first lesson of this course ([link](#)). This parameter holds every string given as argument to the program. For example, if you type the following command line:

```
#> ./program arg1 arg2 arg3
```

NOTE: The “#>” symbol is the prompt of the shell (the characters shown at the start of the command line of any given shell).

the parameter “argc” is equal to “4” and the “argv” parameter is:

```
argv[0] = “./program”  
argv[1] = “arg1”  
argv[2] = “arg2”  
argv[3] = “arg3”
```

From this example, you can deduce “argc” will be always greater or equal than 1, since at least “argv[0]” holds the path and filename used to invoke the executable.

Continuing with our program, there are multiple ways to print a message to the screen. One of them is the function “printf”. You can find more information about it using the “man” command. Anyhow we indicate few basics to let you know how to use it. In our case, since we want to print the message “Hello World!”, the source code should be “printf(“Hello World!\n”);”.

On the one hand, the character “;” is mandatory to indicate the end of a high level source code line. On the other hand, the special combination “\n” indicates a “carry return”. In fact, the message format of “printf” accepts several combinations of characters to include special characters, such as tabulator and carry return, whilst other combinations, that start with “%” let you automatically convert and include to the string other values, such as floats and integers. In fact, in this session you will play with it and we strongly suggest you to search more information about it in the “man” command when you need it.

Since the main function returns an integer value, we have to introduce the code line to do this final step. In this case, as standard, the value “0” means successful execution, whilst non-zero value indicates an error. Therefore, write down “return 0;” after the “printf” code line.

At this point we have finished the development of the body of our program. However, there are still few additional things to be fulfilled in the following steps.

Compilation

This stage of the procedure describes the translation of a file from a given language to another one. Depending on the coding language you need to use a particular compiler. In our case, since we use C language, there are few different options, but to standardize the executables building and behaviour, we suggest to use the “gcc” compiler. For sure, you may find any required information using the “man” command, but we recommend you to read the details of the “-c” and “-o” options.

In our compilation process, we translate from C-code to binary format to be executed. However, there is an intermediate file format a.k.a. “object file”. This format is a content compiled in a similar way than the final program, but with a non-executable format. To perform this step you need to execute the following command line, that generates an output file with the same name than the source code, but with the extension “.o”:

```
#> gcc -c hello.c
```

However, if you have followed the above steps described in this document, you should not be able to successfully execute this command line. You should see a compiler error message indicating something related to the “printf” function. The meaning of this message is you are missing an important component in your code.

In any compiled language, you need to declare the header of the functions you use in the code, before they are invoked. If the function is not implemented before or the code is not even present (e.g. it is in another C-file) the compiler just needs the header declaration to properly parse and perform the first steps of the compilation procedure. To do this in C you need to use the “#include <file.h>” statements at the beginning of your code, where “#include” indicates to the compiler to search the file “file.h” (a.k.a. header file) in a well-known list of directories of “header files” that the compiler is aware of. In case you want to use a particular header file developed by you that is present in your current working directory, then you have to change the “< >” symbols to “ ”. Thus, to solve the problem you currently have, check with the “man” command the header files you need to include for the “printf” function and then reexecute the last command line. You should get no other error or warning message and get an output file called “hello.o” in the same working directory. If so, it has been successfully built.

As you may guess, you have not implemented the “printf” function, but you have used information from a header file that indicates how the function header is. The reason behind this is because the body of “printf” is in an object file provided by the C-language toolkit, a.k.a. C-library (libc). A library is a set of precompiled files (i.e. object files) encapsulated into a single file. The “libc” library comprises a set of object files that include a large pool of functions used in C-coding. Thus, to relate the function call in your code to the implementation of “printf” inside the library you need to perform the final step of the compilation: the linkage. When the compiler performs this final step relates every single object file and library with each other to have a single program file, the executable. To do this, you can execute the following command line:

```
#>gcc -o hello.exe hello.o
```

This command line creates an output binary, called “hello.exe”, after linking the “hello.o” object file with the libc library. It is not necessary to indicate this library in the compilation command line, because it is included by default. **NOTE:** *In Linux-like OS, it is not necessary to put the “.exe” extension to the binary.*

To double check the program has been correctly created, execute the following command line:

```
#> ./hello.exe
```

You should see the message “Hello World!” as result. If so, congrats!!! You did your first C-program!!!

Automate Compilation

The “make” tool let’s you automate compilation of software projects by identifying what are the particular items that should be recompiled, as well as providing additional smart procedures. You can access to the embedded link of the [GNU Make Manual](#), where you will find the whole manual of “make” under different formats.

In this course, we will develop basic scripts to be used with this tool. These scripts, by default, are named “Makefile” or “makefile”, although you can specify other names with a particular flag (-f), as we will see in future Lab sessions. The generic structure of this file is the following:

```
Rule1: prerequisites
<TABULATOR>Actions
Rule2: prerequisites
<TABULATOR>Actions
...
```

The Makefile comprises a list of rules (i.e. a label that identifies a given objective to be compiled by the “make” tool) which has a list of prerequisites (i.e. other rules or files to be fulfilled or checked, respectively). Every rule has a list of actions (i.e. command lines) that will be executed, where every line is started by a tab. This is very important, since if there is a space or another character, rather than a tabulator, an error will be triggered.

Let’s edit our first Makefile. To create a rule to compile the above C code, we can create a rule named “hello”. This rule depends on a specific prerequisite: the “hello.c” source code file. That is, the “make” tool will check this prerequisite (a file in this case) to find out whether there is a need to execute the actions of the rule because either the executable does not exist yet or because the source code has been modified. Otherwise, if there are no modifications in the source code, the actions are not executed because it is not necessary. The “make” tool performs this by comparing the file modification times of the list of prerequisites (if there are files) with the modification time of the file to be generated (by default, the name of the rule). If the timestamp of the prerequisite is later than the modification time of the rule, the “make” tool understands there is a change in the source code and then the actions of this rule need to be executed. In our particular case, the single rule of your first Makefile should be similar to:

```
hello: hello.o
<TABULATOR>gcc -o hello hello.o
hello.o: hello.c
<TABULATOR>gcc -c hello.c
```

To execute this make-script you have to invoke the “make” tool from the command line. When it is executed with no input parameters, it searches (first) the “Makefile” or (then) the “makefile” files in the current working directory. Then, if there are no input parameters, it executes the first rule of the script. Try it yourself by executing “#> make”. If you want to specify a given rule, you have to introduce the rule name as input parameter in the command line, such as “#> make hello”.

Normally, the developers introduce a rule to remove any generated file (for example, executables). In this case, you have to edit the Makefile to introduce the following rule at the end of your current Makefile. Please, leave a blank line above:

```
clean:
<TABULATOR>rm -f hello hello.o
```

As you may see, this rule, named “clean”, has no prerequisites. That is, the rule is always executed when it is called. In this case, it will remove the executable file, called “hello”. Once you have saved the changes in the “Makefile”, try to invoke this rule from the command line: “#> make clean”. You should have the executable “hello” generated before, in order to be removed. The “-f” flag to “rm” indicates not to give an error if the file “hello” does not exist.

Besides, developers normally create a rule called “all” to compile everything. For the convenience of developers, this rule is usually the first one of the script, since this rule is the one executed when there are no rules invoqued from the command line (e.g. “#> make”). The prerequisites of this rule comprises the name of all major rules of the software project. In your case, the name of the rules that generate new executables, like “all: rule1 rule2 ...”. Unlike other rules, the “all” rule has usually no specific

actions, because it is just used to invoke the other rules. In your case, the “all” rule should be as follows:

```
all: hello
```

Currently, it holds a single prerequisite, because we have only one program that can be compiled. As you keep working on this Lab session, you will have to include additional prerequisite names into this “all” rule.

Dealing with input parameters

Now that you are ready to develop basic C programs, we are going to dive into treatment of input parameters from the command line. To do this, let’s copy the source code “hello.c” to a new filename, called “hello2u.c”. Open an editor to work with the new file. The main goal of this new program is to say hello to the name passed by parameter from the command line. For example, if we execute “#> ./hello2u Xavi” we want to see the message “Hello Xavi!”.

You have to remind that the input parameters can be accessed by the “argv” parameter in the source code. Thus, “argv[0]” has the string “./hello2u”, whilst the “argv[1]” comprises “Xavi”. In other words, you have to include “argv[1]” data in the output message.

Exercise 1

*Modify your Makefile to include a new rules to properly compile the new “hello2u.c” source code. Remember to update the “all” and “clean” rules as well. Besides, modify the “printf” function invocation as you may need to print the value of the string variable. To do this, check with the “man” command the “conversion specifier” that you need to use, in conjunction with the “%” character, to include the string provided as input parameter in the command line. **NOTE:** it is important you take into account printf needs the values just after the message, separated by commas.*

Final Indications of C-coding

In this session you have got experience with C-coding and compilation. We strongly suggest to check the C-reference, that can be accessed through this [link](#), in particular to the “[Statements](#)” section. You will be able to see how different parts of C-codes can be developed. It is very likely you will need some of them in future codes to be implemented, but throughout this course you will learn the minimum things that you need to fulfill any exercise proposed from our side.

Exercise 2

Develop a new code, called “calculator.c”. This file implements a calculator that accepts three input parameters: two integer operands and the operator (“add”, “sub”, “mul”, “div” to indicate a sum, subtraction, multiplication, and division, respectively). As a result, the program prints the result to the screen. Obviously, modify the Makefile as required. We provide you the following hints:

- As a first step, assume the operands are integers and positive.
- Use the if-then-else or switch-case statements (both are correct) to implement the code to check what is the operation you want to do.
- Check the “atoi” and “strcmp” functions with the “man” command.
- Check how to use the “printf” function to properly print the result to the screen.

Upload the Deliverable

To save the changes you can use the tar command as follows:

```
#tar czvf session3.tar.gz Makefile *.c
```

Now go to RACO and upload this recently created file to the corresponding session slot.