Data Representation

Computadors – Grau en Ciència i Enginyeria de Dades – 2020-2021 Q2

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

The goal of this session is two-fold. Firstly, it aims at introducing basic background to develop, compile and execute codes in C and C++. To give you additional support for C++ coding, we attach a link to the C++ language reference web site. This first goal also includes basic knowledge to generate an executable from a C/C++ source code.

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 exemple, "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.

Reminder of a C++ Source Code

Let's create a first source code, called *"hello.cc"*. This program will show a "Hello World!" -like message to the screen. Firstly, you have to develop the main function, called "main", which should have the following header:

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

By default, the main function returns an "integer" (a.k.a. "int"). In subsequent lectures you will learn how to receive and process this return value from the shell and scripts. 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. 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 invoque the executable.

Continuing with our program, there are multiple ways to print a message to the screen. In this course, for the sake of deterministic behavior of the system and to ease the learning of new concepts, we will use the combination of "sprintf" and "write" functions. To do so, you have to understand the header of both functions and their functionality:

- **sprintf:** copy a given string (second parameter) to an array of chars (first parameter). It returns the number of bytes that have been copied. For example:

```
char buf[256];
int num;
num = sprintf(buf, "Hello World!\n");
```

- **write:** write to a given destination file (first parameter) the contents of a particular memory address (second parameter), but only a given number of bytes (third parameter). It returns the actual number of bytes written. For example:

```
write(1, buf, num);
```

The above code snippets behave as follows: We copy the string "Hello World!\n" to "buf" (NOTE: the "\n" special character is translated into "carriage return - CR, and line feed - LF"). Then, we write that content to the screen (or the standard output file), indicated by the number "1" as first input parameter of "write" (we will explain in the OS Lesson the meaning of this number).

The body of the main function should end returning an integer value, since it is the return data type of such function. You can simply introduce "return 0;".

Before you compile the source code, you have to complete some requirements. When you use functions, you have to provide some previous declaration of them to the compiler, especially if the functions are developed outside your source code. That is, at the beginning of the file you have to introduce "#include" statements to embed required header files that hold necessary declarations to use those functions.

Exercise 1

Find out what are the missing header files by looking up the manual of the functions used in your code. Edit an "answers.txt" file and indicate what files you have introduced in the code.

Hint: #> man 3 strlen #> man 3 sprintf #> man 2 write

Write what is the meaning of 2 and 3 in the previous "man" commands in your "answers.txt" file.

Your first compilation

To translate the source code into an executable we have to compile. In this course we will use the "g++" compiler installed in the Linux OSes. In this first session we will only generate the executable, but in other sessions you will learn to generate other type of files.

The compiler generates a default named executable ("a.out") if you do not indicate the output filename. To indicate the filename to be created you can use the flag "-o filename" as follows:

```
#> g++  -o  hello  hello.cc
```

This command line invoques the "g++" compiler to compile the source code "hello.cc" to create the executable called "hello".

The powerful make tool

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.cc" 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.cc
←TABULATOR→gcc  -o  hello  hello.cc
```

To execute this make-script you have to invoque 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 invoque 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 invoque 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.

Once you have double checked that your “Makefile” works properly, generate the “hello” program again, and launch it from the command line, as follows: “#> ./hello”. You should see the “Hello World!” message.

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.cc” to a new filename, called “hello2u.cc”. 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 Alex” we want to see the message “Hello Alex!”.

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 “Alex”. In other words, you have to include “argv[1]” data in the output message. To do this, you have to modify the sprintf function call as follows:

```
num = sprintf(buf, "Hello %s!\n", argv[1]);
```

The special character "%s" is substituted by a string and, in particular, by the subsequent comma-separated input parameter of "sprintf".

Exercise 2

Modify your Makefile to include a new rule called "hello2u" to compile the new source code. Remember to update the "all" and "clean" rules as well.

Once you have compiled the new program, you have to execute it introducing an input parameter in the command line, such as "#> ./hello2u Pep". The execution of this command line should trigger the message "Hello Pep!".

Scalar Data Types

Symbols and characters

In the above code, "hello2u.cc", you have shown a message that consists of alphanumeric symbols to the screen. By default, the data read from and sent to the terminal is encoded in ASCII.

Exercise 3

Copy the "hello2u.cc" code to a new file named "hello2uchars.cc". In this new source code you have to add a loop to show every single character of the input parameter from the command line and the respective decimal value of every character. For example, if the character "argv[1][i]" is the character "a" then the decimal value would be 97. Actually, you can check the ASCII code table in the embedded link. To be able to fulfill this code, we give you a few additional hints:

- *The end of the string is delimited by a byte with value zero (0), as well as represented by the special character '\0'.*
- *sprintf can show decimal values and single characters by using "%d" and "%c", respectively. For example,* ***sprintf(buf, "value %d and character %c", 'a', 'a');***
  would copy to the variable "buf" the string "value 97 and character a".
- *Modify your Makefile to include a new rule called "hello2uchars" to compile the new source code. Remember to update the "all" and "clean" rules as well.*

With this Exercise 3 you can understand how characters are encoded when using ASCII and the different meaning when you show an ASCII symbol compared to when you are showing the decimal value of the same byte.

Unicode

Download from (https://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/FilesS3.tar.gz) the support file FilesS3.tar.gz, uncompress, and compile the code in the FilesS3 directory, using

```
#> make
```

Execute the 3 files resulting from the compilation:

```
#> ./utf16-in-c   |   less

…  <look at the output with some detail>

#> ./utf16   |  less

…  <look at the output with some detail>
```

```
#> ./is   |   less

… <look at the output with some detail>
```

Exercise 4

*Explain briefly the output of the "*uft16**" *and "*is*" executables.*

Integers

As discussed in theory lectures, decimal integer numbers may need different data type representations depending on the range of values. In this part of the Lab session we are going to play with different scenarios in which a given number is hold with different data lengths, such as char, short, integer, and long long. Nevertheless, before addressing further exercises, we introduce a new background support:

- "sizeof(data_type)" is a function that returns how many bytes the "data_type" uses.

Exercise 5

Develop a new source code, named "integers.cc". This code reads a number provided as the first argument in the command line. Since the parameter is encoded in ASCII we have to use the "atoi" function (use "man" to find out how to use it) to convert the ASCII string (argv[1]) to an integer. Once the number is assigned to an integer, write it directly to the output. That is, DO NOT use the "sprintf" function to convert the number to ASCII again, as did it in the previous exercises. For example, if we hold the decimal value in an integer variable called "num", we have to do:

```
write(1, &num, sizeof(int));
```

This line invoques the "write" function to write "sizeof(int)" bytes (that is, 4 bytes) to the screen, and the contents of the message is an integer (hold in the memory address "&num") encoded as it is.

To ease the understanding of this exercise, redirect the output of this program (as we introduced in the first Lab session) to write the message to a given file. For example, "#> ./integers > num.dat". Afterwards, we can analyze the contents of this file. However, if you open a text editor or dump the contents directly to the terminal you will see weird characters, because text editors and the terminal process every byte as if they were ASCII codes. To circumvent this problem, you can use the program "xxd". This program shows the contents of any file, introduced as input parameter, at byte-granularity showing the hexadecimal values. Furthermore, as this program writes the hexadecimal values byte by byte, you are able to find out whether the data is stored using big-endian or little-endian in memory.

We strongly suggest you to write a decimal number that you may know what is its hexadecimal representation. For example, the decimal number "1234" is encoded as "04D2" in hexadecimal (i.e. it is the same example that we discussed in the slides of the first Theory lesson).

Exercise 6

Edit the "answers.txt" file and briefly explain what is the output of "xxd" when you execute it to dump the contents generated by your "integers.cc" program. That is, from the output of "xxd", is this system using little-endian or big-endian to save the data?

Exercise 7

Use "integers" to see the output of:

```
#> ./integers  $((0x0a0aa0c3))
```

Without redirecting it into a file. Explain what datatype you are generating in this case. Write your answer in the "answers.txt" file.

Another important point to be analyzed is the difference among data types depending on the range of values. Copy the "integers.cc" source code and name it "inttrunc.cc". In this code, you should remove the write function that directly writes the integer value to the output. Then, introduce few additional code lines in order to assign the integer number to a variable of a shorter data type, for example a character (char). Afterwards write a message (with sprintf-write functions) to the screen to compare both values.

Exercise 8

Edit the "answers.txt" file and briefly explain if there are differences between both values when you run the program with input parameter equal to: a) 27; b) 250; c) 258; and d) 1234.

Floating point

In this final part of the Lab session, we will play with non-integer numbers. To do this, copy the previous source code "integers.cc" and named it "floats.cc". As a first testing step, try to write the float variable directly to the output (as we did it in the previous exercise). Since we have not entered into the details on how to calculate every single bit of a floating point number, we strongly suggest you to write the same number that we used in the slides of the first theory lesson (in particular, slide number 20). That is, assign the number "23.46875" to a float variable and directly write it to the output. Then, inspect the contents with the "xxd" command. We advance you that the floating point representation of this number is "41 BB C0 00" in hexadecimal. Nevertheless, you will see the order of this byte values based on the endianness of this system, as discussed in previous exercises.

Exercise 9

Copy the "floats.cc" source code and name it "floatdiv.cc". The program accepts two numeric input parameters. We can assume both of them are integer numbers that will be the operands of a division. By default, C++ assumes the division of two integer numbers returns an integer result. The developers can force to return a floating-point result by doing a "cast" of every operand. Remember that "cast" means type conversion. It is used by indicating the type to be converted to in parenthesis just before the original variable/value. For example, assuming that "result" is a "float variable", the code "result = (float) num;" means that the value held in "num" is assigned to "result" as a floating-point value. Therefore, to fulfill this exercise, you have to write the result of the division directly to the output, as follows: write(1, &result, sizeof(float));". As before, analyze the output with the "xxd" command.

Finally, although this course does not dive into bit granularity we want to show you how simple operations can involve precision loss problems of floating point data. If you execute your developed "floatdiv" program with "1" and "10" as input parameters, which should result as "0.1", you can actually see the result is "3D CC CC CD". As we know the structure of a "float" (slide 18th of the 1st theory lesson) you may see this number is actually structured as follows:

Sign 1 bit	Exponent (8bits)	Mantissa (23 bits)
0	0 1 1 1 1 0 1 1	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1

Besides, if we apply the formula (slide 20th of the 1st theory lesson) we can calculate, the actual value:

Sign: **positive**

Exponent: $(01111011)_2 = (123)_{10}$ → According to the IEEE format the exponent is $(123 - 127)_{10} = (\mathbf{-4})_{10}$

Mantissa: $(10011001100110011001101)_2 = (\mathbf{0,600000023841857})_{10}$

To sum up, $(2^{(-4)})_{10} * (1 + 0,600000023841857)_{10} = 0,0625 * 1,600000023841857 = \mathbf{0,100000001490116}$ which is actually different from **0.1** ☺.

Compound Data Types

Structures vs union

Develop a simple program (ages-struct.cc) that collects names and ages from the command line, and saves them in a variable of compound type:

```
struct  person_data {
    char name[256];
    int    age;
};
```

After collecting all arguments into the compound variable, a loop should show one person per line, in the form "person i: name-i age-I".

Afterwards, copy the file and rename it to "ages-union.cc". Change the struct to use a union, with the required modifications to properly access manipulate the values.

Exercise 10

Include the output of the execution of this command line, into the "answers.txt" file:

#> ./ages-struct Alex 21 Manel 41 Pep 23

#> ./ages-union Alex 21 Manel 41 Pep 23

Upload the Deliverable

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

```
#tar czvf session3.tar.gz answers.txt Makefile *.cc
```

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