Data Representation

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q2

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

This lab session addresses the variety of data types discussed during the first topic of Theory lectures. In particular, you are going to work with scalar (character, integer, floating point), and aggregate 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.

During the session we ask you to follow detailed instructions on several topics. Try to understand each step indicated, and check with your professor in case you have doubts.

Download the associated examples package: FilesS4.tar.gz <link>

Uncompress the examples on your directory (***NOTE:*** *the symbols "#>" represent the prompt (the header) of the command line in the terminal. That is, you don't have to introduce them*):

```
#> tar  xvf  FilesS4.tar.gz
```

A new directory named S4 should now be in your current directory.

To easy the compilation, we suggest to write a Makefile (as indicated in the previous session) to cover the compilation of all sourcecode of this Lab Session.

Scalar Data Types

Numbers

Open the sourcecode "writeint.c" in a text editor. Understand how we generate an output in binary format. This code reads numbers provided as input parameter in the command line. Since the parameter is encoded in ASCII (https://www.asciitable.com/) we have to use the "atoi" function (use "man" to find out how to use it) to convert the ASCII string (argv[i]) to an integer. Once the number is assigned to an integer, write it directly to the output (use "man" to find out more details about "write"). That is, DO NOT use the "printf" function to convert the number to ASCII again. 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 (first input parameter equals to "1", although in future sessions we will play with other destinations), and the contents of the message is an integer (hold in the memory address "&num") encoded as it is. We also use "cast" for type conversion. It is used by indicating the type to be converted to in parenthesis just before the original variable/value.

If you execute the code as it is "#>./writeint 10" you will see nothing in the screen, because the program writes the binary format of the integer value "10" to the output. Check in the ASCII table (see previous link) what is the meaning.

Exercise 1

*Create a text file called **"answers.txt"** to answer the following questions:*

a) *write the command line you need to execute to redirect the output to a file called "out.dat" instead of the screen. Try to firstly write this single value "10" as input parameter.*
b) *explain your findings about the output obtained from executing the command line "#>xxd out.dat", in terms of endianness (check theory slides) and number of bytes the file comprises. The command "xxd" was used in the Lab Session 2.*
c) *write the command line to append more values to the "out.dat" file executing new invocations to "./writeint". That is, the data included in previous command line executions are preserved in the file. You can do it using the values that you wish and using multiple input parameters in a single invocation.*

NOTE: you can also specify values represented in hexadecimal directly from the command line using the format "$((0xNUM))". For example, "$((0x12AB))" would be the hexadecimal "0x12AB".

Exercise 2

*Copy the file "writeint.c" and name it **"writebin.c"**. Modify the code in such a way the first input parameter indicates the type of data you want to write out following binary format. For example, "char", "short", "int", "long", "longlong". Hints: check the "man atoi" to see how to convert "long" and "longlong" values (others are covered by "atoi"); check "man strcmp" to check the string of the first input parameter. You can validate your code executing command lines like the previous exercise.*

Open the sourcecode "readchar.c" in a text editor. It is similar to the "writeint.c", but simpler. The main goal of this code is to read chars from the standard input and print to the screen the read values, depending on the data type. You can execute this code as it is ("#>./readchar") and then you can write with the keyboard and press "Return" and you will see the values of every single byte. Press "Ctrl+C" to finish the execution.

The code uses "read" (use "man" to find out more details) to read from the standard input (first input equals to "0", although in future sessions we will play with other sources).

Exercise 3

*Indicate in the **"answers.txt"** file the command line you need to execute to redirect the standard input to read from a file called "out.dat" instead of from the keyboard.*

Exercise 4

*Copy the file "readchar.c" and name it **"readbin.c"**. Modify the code in a similar way than the exercise 2. That is, it accepts an input parameter to indicate the type of data you want to read. We strongly suggest to use the "out.dat" file as data input to validate this exercise.*

Exercise 5

*Indicate in the **"answers.txt"** file your findings when you write a value out-of-range to the "out.dat" file. As well as, indicate your findings when you read a value out-of-range from the "out-dat". To fulfill this exercise you should use your codes. In case you were not able to fulfill them, you can use the sourcodes attached to this session as alternative.*

Symbols and characters

We have included a sourcecode called "utf8coding.c". Analyse the code in a text editor, but understanding the processing of the UTF-8 characters is beyond the goals of this exercise. The code writes the UTF-8 character set from the values read in the standard input. Execute this code using "#>./utf8coding -?" to see the help to know how to invoke it.

Exercise 6

Use the "./writeint" code to write the values 100000 and 127829 to a file called "out.dat". Then execute the "utf8coding" program to read from this file. Just do it in a regular mode (without flags) and in verbose mode (with the "-v" flag). Indicate in the ***"answers.txt"*** *file whether both symbols require the same number of bytes to be represented or not, as well as why we need the UFT-8 representation instead of ASCII to represent those symbols.*

If you want to try other symbols, you can go to the following link: https://www.utf8icons.com/. In every symbol you will see the decimal value you have to introduce (see "Symbol Information Table" of any symbol or character).

Floating point

The "writefloat.c" attached code is similar to the "writeint.c" file, but writing a float in binary format. In particular, we assign the number "23.46875" to a float variable and directly write it to the output. Like done before, use an output file, such as "out.dat" to save the data. Following the examples seen in theory and exercises 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 7

Copy the "writefloat.c" source code and name it ***"floatdiv.c"****. The program accepts two integer number, performs a division between them and write the result in binary to the output. 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. 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.*

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" (check theory slides) you may see this number is actually structured as follows:

Sign 1 bit	Exponent (8bits)	Mantissa (23 bits)
0	01111011	10011001100110011001101

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: $(10011001100110011001101)_2$ = (**0,600000023841857**)$_{10}$

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

Compound Data Types

Structures vs union

Exercise 8

*Develop a simple program **"ages-struct.c"** 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;
};
```

First, there should be a loop to collect all arguments (argv) into the compound variable, and then a second loop to show one person per line, in the form "person i: name-i age-I".

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

Exercise 9

Execute the following command lines:

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

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

*Include the output of the execution of this command line, into the **"answers.txt"** file, as well as why the output of ages-union is different than ages-struct.*

Upload the Deliverable

*To save the outputs **(all bold filenames highlighet in the document)** you can use the tar command as follows:*

```
#tar czvf session4.tar.gz answers.txt Makefile *.c
```

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