KU LEUVEN

GROUP T LEUVEN CAMPUS

Operating Systems

Programming with C: lab 1

Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse
Lab coaches:

- Ludo Bruynseels
- Toon Dehaene
- Jonas Verbruggen

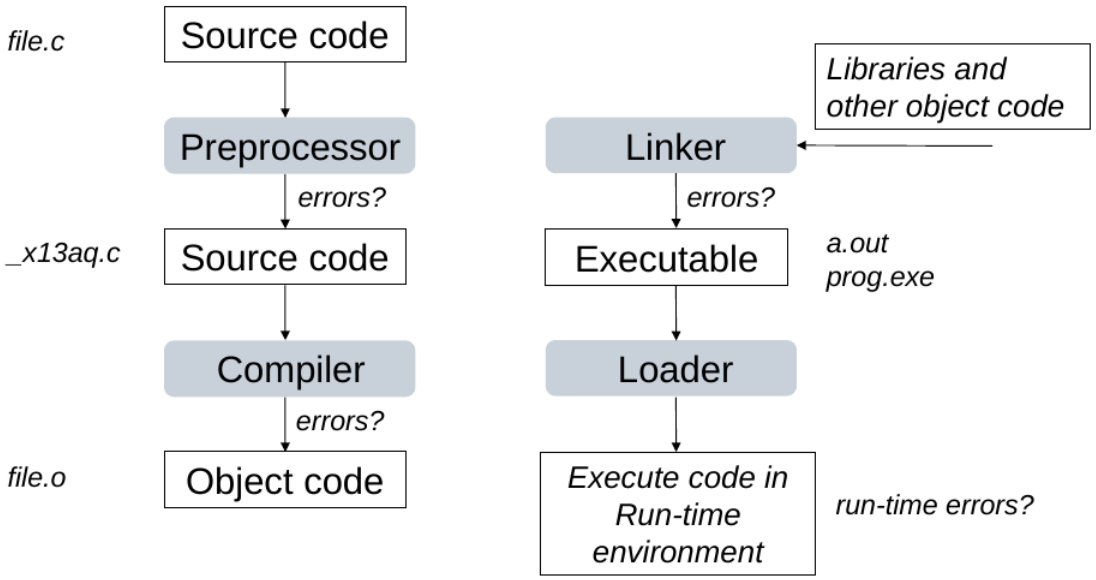
1. Introduction: Setting up your development environment

In this lab we will emphasis the most important GIT commands. We want to show that Git from the command line is not difficult and very powerful. Mastering these commands will help you debug your git repository when you get confused by git interfaces in the IDE.
We will also set up a repository in the cloud: github. The gitlab server at GroepT which was used in the past cannot be used anymore. Therefore, we move to github.
A third tool that we need to know is GCC, the compiler/linker.
And to save us from typing the same commands over and over we use the make tool to script the build process.

2. Introduction to GCC

The GNU C compiler collection (GCC) is used for **all** programming exercises of this course. Gcc is one of the most used C/C++ compilers, not only on Linux where it comes pre-installed, but on many other platforms as well.
GCC is more than only a set of compilers: it's really a collection of libraries and tools like gprof, gcov, gdb, etc. The entire GCC is open-source. Visit the GCC home page[1] for more information.
C program code must go through a number of processing stages before it can be loaded and run. First, C code must be preprocessed. Next, the compiler transforms the code to object code which then can be linked into an executable. The full processing chain is depicted below.

1 gcc.gnu.org



More details on the different processing steps will be given during this course. Hence, be patient. For now, it suffices to know that given a text file 'main.c', an executable can be compiled on the command-line prompt ($) with the command :

```
$ gcc main.c
```

After running this command, there are two possible results. The first - and probably the most likely - result is that an error occurs, typically because of compiler/linker errors. You must first solve all errors before you can retry. Hopefully, after some retries, the other possible result pops up on your screen: a successfully compiled program. In that case, you should find an executable file 'a.out' in the same directory. Check if this file has executable permission and if needed, modify the permissions of this file. Finally, you can execute the program with the command './a.out'.

```
$ chmod +x a.out
$ ./a.out
```

There are two more options worth knowing before you start programming.

- First, the '-o' option allows you to give a more meaningful name to the executable file. For example, 'gcc main.c -o helloworld' creates an executable file called 'helloworld', assuming there were no errors during compilation.
- Second, the '-Wall' option turns on many warning flags used by gcc to check for potential mistakes. Do **not** neglect warnings because in many cases warnings are forerunners to some future problems.

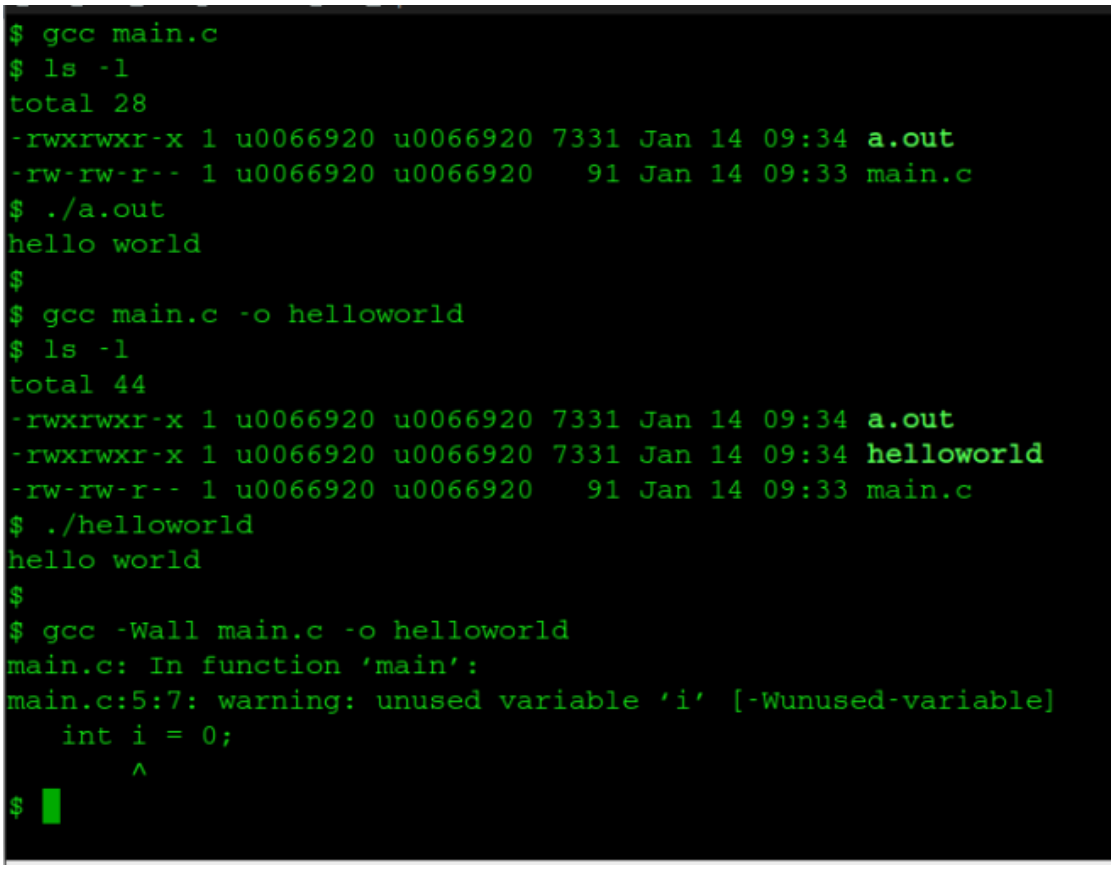
```
$ gcc -Wall main.c -o helloworld
```

Therefore, many programmers and companies treat warnings at the same level as errors. We also insist that you **always** compile with '-Wall' and that you only consider the program successfully compiled when **no** more warning messages show up. The 'all' in

'-Wall' suggests that all warnings will be captured, but that's **not** true. In the man pages of 'gcc' it is explained which warning flags are set and which aren't set with the '-Wall' option.

```
$ man gcc
```

The most important described commands are illustrated in the screenshot below. Notice the difference between using '-Wall' or not.



General hint: There exist very nice tutorials for the C language and for the C standard library with example code for many library calls. For example:

https://www.tutorialspoint.com/cprogramming/index.htm
https://www.tutorialspoint.com/c_standard_library/index.htm

3. Exercise 2: datatype size, type qualifiers

Create a directory ~/osc/clab1gcc/
In this directory create all the sourcefiles you need. Do not forget a Makefile.

The memory size of the built-in data types int, float, etc. is system dependent.

- Write a program that prints the memory size of int, float, double, void, and pointer using the `sizeof()` library function.

- What happens to the sizes when you use the type qualifiers `short` and `long`?
- Find out if a char is an unsigned or signed data type on the system you use.
- Commit and push all your code to the git repo:
  - $ git status
  - $ git add
  - $ git commit
  - $ git push

3. Exercise 3: strings

All header files and source files for this exercise can be placed in the current working directory. Just make a new target in the Makefile.
There is no built-in data type 'string' in C. As a matter of fact, strings in C are defined as null-terminated arrays of char. For example:

```
char name[4] = “luc”;
```

The array 'name' defines a null-terminated string which means that name[0] = 'l', name[1]='u', name[2]='c' and name[3]='\0'. The termination '\0' is very important as all string manipulation functions will use this character to detect the end of the string. Also notice the difference between single and double quotes: 'c' indicates the constant char c, but “c” defines a constant string containing only one character being c. But if you store the string “c” you need a char array of length at least 2 because also the '\0' char must be stored to terminate the string.
Fortunately, to ease the pain of not having a built-in data type string, the C standard library provides an entire set of string manipulation functions. The target of this exercise is to explore a number of these functions.
Write a program that declares the string variables *first*, *second*, *name* and *str* both of some maximum length MAX.

- Use `scanf` to read your first and second name in *first* and *second*, respectively.
- Convert your second name to all upper case chars and store the result in *str*.

You can convert between upper case chars and lower case chars without using the library functions. Find the ASCII values for digits and lower/upper case chars. Make your own functions to test for digit, upper/lower case chars.

- Apply `strcmp` on *second* and *str* and make sure you understand the result. Is there another compare function ignoring the case of chars?
- Concatenate your first and second name into *name*; be aware that there is an unsafe and a safe string copy and concatenation function. Print the result.
- Read your birth year in some int *year*.
- Concatenate *first*, *second* and *year* into *name*. This time, use `snprintf` to do the concatenation. Notice that there is also an unsafe version of `snprintf` being `sprintf`.
- Finally, use `sscanf` to read from *name* the *first*, *second* and *year* values again. Print the result on screen to check the result.

When you are done, commit everything in your local git repository and push it to github.