NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

Remembering the GCC toolchain and other tools

LAB 0

NOTE: This lab revisits something you've done in the course "Arquitetura de Computadores" last semester, in labs 0, 1, 2, 3 and 4 (go reread those). You should be able to follow this lab with no external help. However, bring all your questions to classes, so that you become prepared for this course.

You need a working Unix or Linux system computer (or VM). You can use the one provided for FSO (at `https://drive.google.com/file/d/1yScFQZEPm0bva4bihk4pC1IN5qJdBRIV/` with login `fso`, password `fso`). You can try to use the one from AC, or, if using an Apple computer, you can try to use the installed Unix variant, the MacOS.

1. Lab objectives

The C development toolchain in the Linux systems, is usually built from the GNU C Compiler (`gcc`) and related tools like GNU binutils. Those support compiling and linking programs and also important tasks like debugging (`gdb`), testing and inspecting the executable code.

This Lab will allow you to review each component's role, how those can help you to pinpoint the sources of errors in a much more simple and accurate way than simply screaming "Prof, I have a compile error!" or "Why my program aborts?". This knowledge can save you lots of hours (or days) when developing your programs in FSO and any other course! A professional programmer must learn the tools of its craft that make his/her work easier.

2. The GCC toolchain

The GCC toolchain pipeline is depicted in Fig. 1 below, including preprocessor, compiler assembler and linker. Usually, you do not see those steps, as most of the time, you just need to call the C compiler (`cc`, also called `gcc`) and it will do everything (without producing intermediate files).

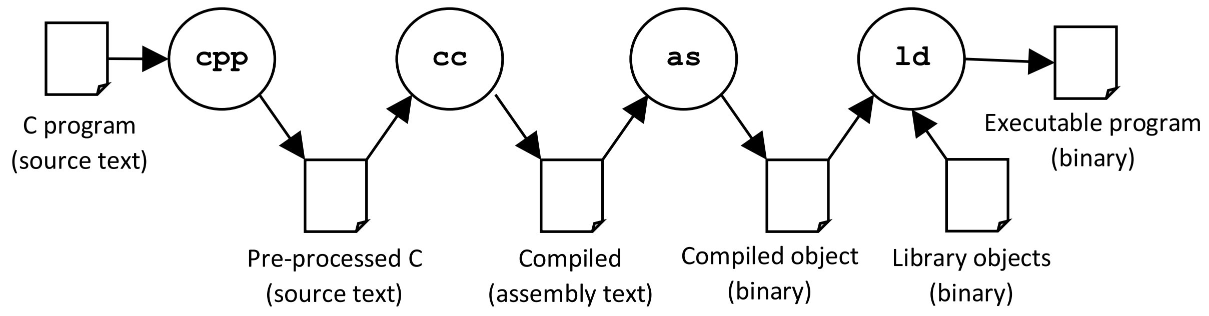

Figure 1. The C compiler toolchain

2.1. The C preprocessor

The **cpp** is the C pre-processor. It takes care of every line that begins with `#`, interprets it, replacing the original text and producing a file which contains "pure" C. For example, if you have an `#include` in your program, the C pre-processor reads the referred file into your own program.

You can tell **cc** to stop after preprocessing the file using the `-E` option (or flag): it will printout the text, as changed by the pre-processor, to the terminal screen. If you wish, you can store the text in a file using the **>** shell redirection operator. Example, using the supplied `lab0.c` file:

```
cc -E lab0.c > lab0-preprocessed.c                                  (1)
```

Now look at the contents of `lab0-preprocessed.c`. It will probably have around 1900 lines, and your program, albeit modified, will be at the end. See how the original **`atoi`** has now changed: remember, a `#define` is just a text replacement operation!

Now comment out the `#define` and the `#include` lines (you can do that with two slashes, `//`) and execute (1) again. You will not get any error, as you have not compiled this code yet. Look once more at the contents of `lab0-preprocessed.c`. The preprocessor has just included a few lines. It's not the preprocessor job to complain if you do not insert the adequate includes or definitions!

Now, let's see if the compiler is going to complain about that…

2.2. The C compiler

The C compiler is usually called **cc** in the Unix like systems (the `gcc`, for GNU's `cc`, is the same program). You can compile `lab0.c` to produce the executable file `lab0` with the following command:

```
cc -o lab0  lab0.c                                                  (2)
```

That will be the same as compiling the previous `lab0-preprocessed.c`. Why?

Now, you will get a warning, something like "`warning: implicit declaration of function 'atoi'`" and an error, which will be similar to "`error: 'MY_NUMBER' undeclared (first use in this function)`". Errors are fatal: the compiler simply cannot generate code if it doesn't know what `MY_NUMBER` is. Warnings tell you to pay attention: something may be wrong, but the compilation can continue. Remember that the C compiler can produce the code calling `atoi` **as you have written**, but **without** verifying if that call is correct, as you didn't include `stdlib.h` . **Never ignore warnings** as those are usually wrong things that will give you problems later! You can use -Werror to turn warnings into errors, and you can even use option `-Wall` to turn all the warnings on and check all of them:

```
cc -Wall -o lab0 lab0.c                                             (3)
```

Let's remove the // to reinstate our preprocessor directives and compile again, but just until the assembly representation of the code produced by the compiler. For that use the -`S` option:

```
cc -S lab0.c                                                      (4)
```

You now have the assembly of the code produced by the compiler from your C code in the file `lab0.s`. Look at it with any text editor. You can find the `main` function but `printf` and `atoi` are not there. Why?

You can also ask the compiler to just compile your file, producing the binary machine code of your code (instead of the assembly) by calling:

```
cc -c lab0.c                                                        (5)
```

You should find that output in a file `lab0.o` (.o from object). The command `objdump` can be used to disassembly any object or executable file. You can obtain the assembly representation of the machine code in this last file by doing:

```
objdump -d  lab0.o                                                  (6)
```

The C compiler has other options including the ones that make more verifications and help the testing and debugging of your program. We will talk about those below.

2.4. The `ld` linker

The linker binds together machine code modules (object files), from several source files and libraries, producing the executable program. This is usually called by the compiler to link your object with the standard C library, like in the command (2).

Sometimes you will see errors that are not from the compiler, but from the linker and are a sign that you are using functions or variables that are not in your program or in the libraries. Check your identifiers names, as you may have misspelled some name, or something is missing.

As an example, change the name of `printf` to *`print`*. That function is not defined in your program nor in the C library. You will see a compiler warning and a **linker error** similar to: “`undefined reference to ` + "`" + `print'`”.

3. Debugging

All languages include a tool to help in the debugging process. Some kind of debugger. If using an IDE, an interface to that tool is usually also integrated in the IDE. You probably have already used that for your Java programs in the Eclipse, VSCode, InteliJ, etc. If not, look for it and start using it!

The common debugger for GCC toolchain is the `gdb` or some of its interfaces like `gdbtui` (the same as `gdb -tui`) or the ones offered by the IDEs. You can find debugger interfaces in Eclipse, VSCode, CLion, Geany, etc.

Other tools and some compiler options, make some verifications to your code and pinpoint potential problems or uncommon practices. Some tools work on your source code and others during the program execution. Remember `cppcheck`, `valgrind` and the compiler options like `–fsanitize=address`. **Please redo the lab 4 from AC classes about debugging to correct all the problems in `simples.c`**. In the future, you should be able to use a debugger, the compiler options or `cppcheck`/`valgrind` in your daily work.

4. make

When projects are a bit bigger, we start using project managers that help the compilation phase, and allow several build configurations (debug, testing or release). The simplest is the standard `make` command that uses the definition in a file named `Makefile`. We provide one with this lab code. Check the example and read the bibliography.

5. More information

- Don´t forget that manual pages for tools, commands, and C functions are available using the command **`man`**. Example, try: “`man make`” . These documents and others can be found using Google or other search engines (be careful, as some time you can get lost in irrelevant documents and even find information with errors).
- Read from the FSO recommended book, the lab tutorial chapter available at http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf. Pay special attention to the section on Makefiles.
- Look at online free resources about C, such as https://www.geeksforgeeks.org/c-programming-language/, https://www.w3schools.com/c/, https://www.programiz.com/c-programming/