FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
DEPARTAMENTO DE INFORMÁTICA

Introduction to the GCC toolchain
(Lab-01 – September 2019)

1. Lab objective

The C development toolchain is built around the `gcc` (Gnu C Compiler) tools, which include a pre-processor, a compiler, an assembler and a linker, and the `gdb` debugger (which can be used with a GUI frontend, `ddd`).

This Lab will allow you to understand each component's role, when creating an executable program from a C source file, and pinpoint the sources of errors in a much more accurate way than simply stating "look, I got a compilation error!".

2. The GCC toolchain

The GCC toolchain is depicted in Fig. 1 below.

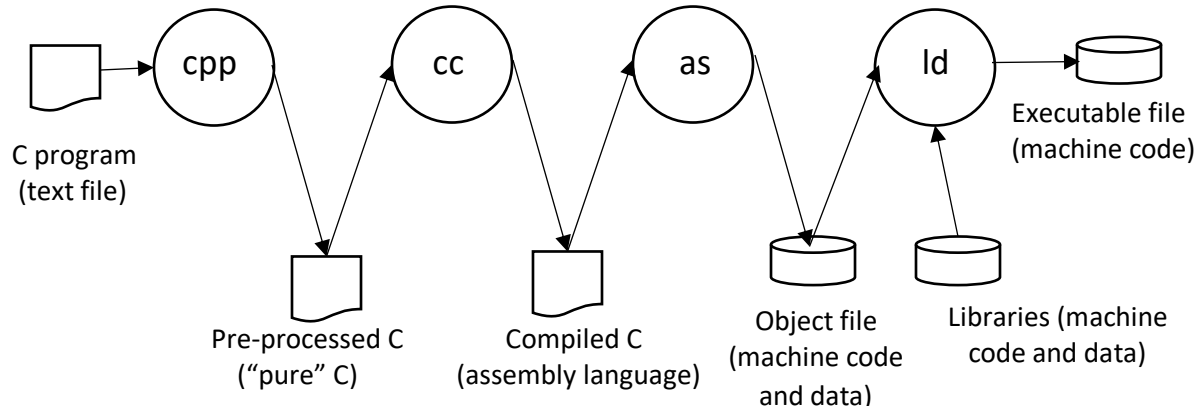

Figure 1. The C compiler toolchain

1.1. The C preprocessor

The cpp is the C preprocessor; it takes care of every line that begins with `#`, and pre-processes it, changing 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 a file and merges its contents into your program (which may become very large, compared to the initial one).

You can tell `gcc` to stop after preprocessing the file using the `-E` 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 `>` bash redirection operator. Try, using the supplied `lab-01.c` file,

```
$ gcc -E lab-01.c > myPreprocessedLab-01.txt                    (1)
```

Now look at the contents of `myPreprocessedLab-01.txt`; 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: you see, a `#define` is just a text replacement operation.

```
#include <stdio.h>
#include <stdlib.h>

#define MY_NUMBER "123"

int main(int argc, char *argv[])
{
	printf("And my number is... %d\n", atoi(MY_NUMBER));
	return 0;
}
```

Listing 1. The lab-01 program

Now comment out the `#define` - you can do that with two slashes, `//` and execute (1) again: you will not get an error, as the preprocessor has no way to "know" that it's its job to replace the word `MY_NUMBER`. Now, let's see if the compiler is going to complain about that… but before, let's also comment out the `#include <stdlib.h>` line, and execute (1). Look once more at the contents of `myPreprocessedLab-01.txt`; the preprocessor has just included a few lines… and it gave no error! The idea is: it's not the preprocessor job to complain if you do not insert the adequate includes or definitions.

1.2. The C compiler

Now, let's invoke `gcc` allowing it to proceed to the end of the compilation fase; you can do it with the `c` flag:

```
$ gcc -c lab-01.c                                               (2)
```

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)`". Warnings tell you to pay attention: something may no be exactly what you intended, but the compilation does continue. Errors, on the other hand, are fatal: the compilation process halts. The compiler simply cannot generate code if it doesn't know what `MY_NUMBER` is.

Let's correct that, removing the // to reinstate our `#define` line, and compile again… but now using an `S` for the flag:

```
$ gcc -S lab-01.c                                                    (3)
```

You still get a warning "warning: implicit declaration of function 'atoi'" but no error. If you look at the files in your working directory you can see that a new file, named `lab-01.s`, has been created. That's the assembly code produced by the compiler.

If you're using a 64-bit architecture (as we have in the Labs) you will see that the interesting part looks like:

```
subq    $16, %rsp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movl    $.LC0, %edi
call    atoi
movl    %eax, %esi
movl    $.LC1, %edi
movl    $0, %eax
call    printf
movl    $0, %eax
leave
```

But if you're using a 32-bit architecture, it will look like this:

```
...
subl    $16, %esp
pushl   $.LC0
call    atoi
addl    $16, %esp
...
```

The interesting thing is that the 64-bit architecture stores the pointer to the "123" string on the `edi` register (and a zero on `eax`) and calls the `atoi` function, while the 32-bit version stores the pointer on the stack and then calls `atoi` (and that's what you have probably learnt in the "Arquitectura de Computadores" course last semester). So you see, things change: the 64-bit architecture has more registers, so the new compilation strategies speed things up by not using the stack.

Now let's do something really stupid 😊: let's modify our `atoi(MY_NUMBER)` function to read `atoi(MY_NUMBER, "456")` and compile again using (3) [remember that the `#include` lines are still commented out]. No error! But the **`atoi`** function only has one argument! Let's look at the code:

```
subq    $16, %rsp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movl    $.LC0, %esi
movl    $.LC1, %edi
movl    $0, %eax
call    atoi
...
```

That is, the compiler does not know that `atoi` is supposed to have just one argument, and blindly follows our own code, using 2 arguments. Will it run? Let's try to generate an executable program:

```
$ gcc lab-01.c                                                    (4)
```

It still complains with a warning, but the executable file (program) is there: `a.out` [remember, that's the name of an executable by default, i.e., if you do not choose a different name].

Let's run it:

```
$ ./a.out                                                         (5)
And my number is... 123
```

Yes! It runs! But the program is wrong… here it had no other (side) effects, but in a more complex program it could be fatal and take hours before we pinpoint the error.

So let's reinstate the `#include`, removing the comments, and compile again using (4)

```
$ gcc lab-01.c
```

Now, we got an error: "`error: too many arguments to function 'atoi'`"; that is, the include files contain the *prototypes* of the standard functions and with them the compiler can check if the programmer is using the functions properly; the error message also shows:

```
/usr/include/stdlib.h:147:12: note: declared here
 extern int atoi (const char *__nptr)
```

Therefore… never use a function without its companion includes (just run `man atoi` – or any other function – to see what must be included), and try to get a program to compile with no warnings – and, to test for "every" possible warning, you should use `-Wall`.

If the compilation succeeds and the executable file is generated, all the other files are deleted.

1.3. The `as` assembler

We will not discuss the assembler here, because we will not need it as we will not write assembly modules nor write inline assembly code in the middle of a C program. `as` will take the assembly source code produced by the compiler and generate an *object file*, containing machine code and "binary" data.

1.4. The `ld` linker

The linker binds together machine code (object) modules and these may be user-developed or C-library modules. For example, in old GCC versions the following program,

```
#include <stdio.h>
#include <math.h>
int main(int argc, char *argv[]) {
      printf("And my number is... %f\n", sin(3.14159));
      return 0;
}
```

Listing 2. The lab-02 program

had to be compiled with **$** `gcc lab-02.c -lm` in order for the code of the `sin` function, which is stored in the `libm.so` file, could be included with the executable. Today, the mathematical functions are linked by default, if necessary, in some cases (see further down for a case where you need `-lm`).

A common mistake is to spell `main` incorrectly, e.g., writing `Main`. If you compile a program with such a mistake, you'll get something like:

```
...
undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Note that this is **not a compilation error**, but a linker error, as shown by the `ld` in the previous line above.

3. Developing large(r) programs

If you are developing a large(r) program than those shown in Listings 1 and 2, it helps if you separate your design into multiple *modules*, and code them into separate files. As a simple example, let's introduce in Listing 3 the two separate files that, together, implement a new program.

```
#include <math.h>

double mySin(double a) {
      return sin(a);
}
```

(a) The mySin.c file

```
#include <stdio.h>

int main(int argc, char *argv[]) {
      printf("And my number is... %f\n", mySin(3.14159));
      return 0;
}
```

(b) The myMain.c file

Listing 3. The lab-03 multi-file program

3.1. Compiling everything in just one step

You may compile the program using

```
$ gcc -Wall -o lab-03 mySin.c myMain.c                    (6)
```

We have added the already mentioned `-Wall` flag, and we introduce the `-o` flag, to specify the executable name (we don't want to use the default `a.out` name any more). We get the following messages:

```
myMain.c:5:37: warning: implicit declaration of function 'mySin' …
… format '%f' expects argument of type 'double', but arg… 2 has type 'int'
…
mySin.c:(.text+0x1b): undefined reference to `sin'
collect2: error: ld returned 1 exit status
```

Why is the compiler complaining about `mySin` type and arguments? After all, `mySin.c` is there and it could check it; but no. The compiler treats each file as a separate unit and does not carry information around from one file to the other. The other thing is: if the compiler does not know about a type, it assumes an integer – therefore, it raises a warning on the `%f` of the `printf`.

There's also a linking error, but we'll look for a solution after we fixed the warnings problem: one step at a time is the best way to learn and understand what you're doing!

So, we need to add a line with the format of the `mySin` function (as you know, we call it the function prototype):

```
#include <stdio.h>

double mySin( double x );

int main(int argc, char *argv[]) {
      printf("And my number is... %f\n", mySin(3.14159));
      return 0;
}
```

Listing 4. The correct myMain.c file

Compiling it again, as in (6)

```
$ gcc -Wall -o lab-03 mySin.c myMain.c
```

fixes the warning problem, but the **linking error** is still there: the linker cannot find the code for the sin function, to build the executable file

```
/tmp/ccatluMi.o: In function `mySin':
mySin.c:(.text+0x1b): undefined reference to `sin'
collect2: error: ld returned 1 exit status
```

Notice that GCC creates temporary files in the `/tmp` directory during the process, deleting them at the end. To fix the linking error we need to add a flag to say to the linker that it should look in the `libm.so` file for the sin code (the `libm.so` file is in `/usr/lib` directory).

```
$ gcc -Wall -o lab-03 mySin.c myMain.c -lm                    (7)
```

So, why didn't we had to add the flag before, when everything was in a single file? I am not 100% sure, so I will not post it here; if you really want to know the details, dig further…

3.2. Compiling each file separately, and then link them together

You may compile each file separately using

```
$ gcc -Wall -c mySin.c                                        (8a)
$ gcc -Wall -c myMain.c                                       (8b)
```

and then link them together with

```
$ gcc -o lab-03 mySin.o myMain.o -lm                          (8c)
```

Notice that each compilation step leaves out a machine code (object) file with the same name as the file being compiled and a `.o` extension. Then, we bind them together in the linking step to create the executable.

Separate compilation is heavily used in large programs, where it helps if we design them as separate modules, each one with at least one `.c` file, where we keep functions and variable declarations that are closely related with each other. It also helps debugging and correcting mistakes, because we deal with just one file (or, at least, one file at a time).

3.3. Separating modules even further: information hidding

A fundamental principle of program design is that any "piece" of code should "know" only the minimum (of code and data) that is required for its operation. Every other piece (functions, variables) should be hidden from it.

Listing 4, below, is a very simple example on how to achieve information hiding in C. It works as follows: the first `printf` will execute the `mySin` function before printing its return value; but the first time `mySin` is executed the value of `BypassMySin` is one (because it was initialized to that value), therefore the value zero is returned and printed. But then `BypassMySin` is changed in the `main` module to zero and, when the second `printf` is executed and `mySin` executed the second time, it finds the value of `BypassMySin` to be zero, and returns the value of the `sin` of 3.141519/4 (45^0 angle) to the `printf`.

```
#include <math.h>

int BypassMySin= 1;

double mySin(double a) {
      if (BypassMySin)
            return 0.0;
      else
            return sin(a);
}
```

(a) The mySin-02.c file

```
#include <stdio.h>

extern int BypassMySin;

double mySin(double a);

int main(int argc, char *argv[]) {
      printf("And my number is... %f\n", mySin(3.14159/4));
      BypassMySin= 0;
      printf("And my number is... %f\n", mySin(3.14159/4));
      return 0;
}
```

(b) The myMain-02.c file

Listing 4. More information-hiding

To compile, we'll use (7)

```
$ gcc -Wall -o lab-04 mySin-02.c myMain-02.c -lm
```

And running it we get, as expected:

```
$ ./lab-04
And my number is... 0.000000
And my number is... 0.707106
```

There's nothing new about the declaration of `BypassMySin` in the `mySin` function: it's just an ordinary, initialized, global variable. But the `extern int BypassMySin` is something new: it informs the compiler that in another module there is an integer variable named `BypassMySin`, and that variable will be accessed by this module.

Now let's do a fairly common beginner's mistake: forgetting to type the `external` keyword, and initializing the `BypassMySin` variable in more than one module.

```
#include <stdio.h>

int BypassMySin = 1;

double mySin(double a);

int main(int argc, char *argv[]) {
      printf("And my number is... %f\n", mySin(3.14159/4));
      BypassMySin= 0;
      printf("And my number is... %f\n", mySin(3.14159/4));
      return 0;
}
```

Listing 5. The myMain-03.c program

Now, compiling does not signal any mistake, **but the linker does**:

```
$ gcc -Wall -o lab-05 mySin-02.c myMain-03.c -lm

/tmp/ccpWCGw4.o:(.data+0x0): multiple definition of `BypassMySin'
/tmp/ccofQEYD.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

This ends our Lab, and our brief introduction to the GCC toolchain: preprocessor, compiler, assembler and linker. Now, you:

- Must read, from the FSO recommended book, the lab tutorial chapter available on http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf with special attention to the section on **`makefiles`**. (We have supplied a Makefile for the lab-03)
- Should get a copy of the most famous C book: C Programming Language, 2nd Edition, by Brian W. Kernighan and Dennis M. Ritchie.
- And look for online free resources such as http://www.cprogramming.com/ and http://publications.gbdirect.co.uk/c_book/ (these two references were obtained from AC classes, they were probably gathered by Prof. Vitor Duarte).

Finally: if you find any error, mistake, typo, etc., please report it to poral@fct.unl.pt. Happy reading!