# Lecture 5: C Program Compilation

## A simple C program

For this lecture, we will learn how to write, compile, and run a very basic C program, and we will discuss the steps that are involved in creating the executable. The following C program prints out the text “Hello world!” to the screen:

```
/*
 * File: hello.c
 * -------------
 * This simple C program prints out the text "Hello world!".
 *
 * COS315 Operating Systems
 * 10 Mar 2003
 *
 */
#include<stdio.h>

int main(void) {
  printf("Hello world!\n");
}
```

To compile this program, we will be using the `gcc` compiler in Linux, which stands for “Gnu Compiler Collection”, and it is used as follows:

```
$ gcc hello.c
```

This creates an executable called `a.out`. To run the executable, you would type

```
$ ./a.out
Hello world!
```

If we wanted to create an executable that is named something other than `a.out`, we would use the `-o` option in the form

```
$ gcc -o hello hello.c
```

and we could run the `hello` executable with

```
$ ./hello
Hello world!
```

In what follows we will go over the details of what actually happens when you invoke the C compiler `gcc`.

# The C compilation model

When you invoked the `gcc` compiler, a series of steps were performed in order to generate an executable from the text you entered as your C program. These steps are as follows

$$\begin{tikzcd}
\text{Source Code (hello.c)} \arrow[d] \\
\boxed{\text{Preprocessor}} \arrow[d] \\
\boxed{\text{Compiler}} \arrow[d] \\
\text{Assembly Code} \arrow[d] \\
\boxed{\text{Assembler}} \arrow[d] \\
\text{Object Code (hello.o)} \\
+ \\
\text{Libraries} \arrow[d] \\
\boxed{\text{Linker}} \arrow[d] \\
\text{Executable (a.out or hello)}
\end{tikzcd}$$

## The preprocessor

The main function of the C preprocessor is to remove comments from the source code and interpret *preprocessor directives* which are given by the statements that begin with `#`. In our simple `hello.c` code, the preprocessor would strip the source of the comments contained within the `/*...*/` and would include the file called `stdio.h`, which contains the standard input/output functions that are usually called within a C program. The `#include` statement can either be called with

```
#include<file.h>
```

or with

```
#include ‘‘file.h’’
```

The first method tells the preprocessor to look for the file in the standard include directories, which for Linux are in `/usr/include`. The second method, which uses the quotes, tells the preprocessor that the file to be included is in the local directory. We will go into more detail on different preprocessor directives as we look at different facets of the C programming language.

## Compiling and Assembling

Once the C preprocessor has stripped the source code of the comments and expanded the preprocessor directives given by the lines that begin with # , the compiler translates the C code into assembly language, which is a machine level code that contains instructions that manipulate the memory and processor directly, in a layer beneath the operating system. Usually, you do not need to see the assembly code. But you can create the assembly code with

```
$ gcc -S hello.c
```

This will create a file called `hello.s`, which looks like

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!\n"
        .text
        .align 2
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

This file contains machine-level instructions like `pushl` and `movl`, which we will not go over in much detail. What is important to understand is that the compiler is directly responsible for converting C syntax into this machine level code.

You do not usually see this level of compilation. Instead, you see what is known as the object code. The compiler creates the assembly code and converts the machine-level instructions into binary code. You can create object code from a C source with

```
$ gcc -c hello.c
```

This creates a binary file called `hello.o` that cannot be viewed with a text viewer.

## Linking

The object file `hello.o` contains a binary version of the machine language that was created from your source code `hello.c`. In order to create the executable `hello` or `a.out`, you need to use the linker to process your main function and any possible input arguments you might use, and link your program with other programs that contain functions that your program uses. In this very simple example, we used the `printf` function. The `printf` function is a standard function that is provided by the C compiler that your current object file knows nothing about. In order to use this function, we need to use the linker in order to link our program with the precompiled libraries provided to us by the C compiler. The linker links other precompiled object files or libraries together and creates the executable `hello`. When you type

```
$ gcc -o hello hello.c
```

the `gcc` compiler creates an object file and does the linking for you. However, when you use the `-c` option, you create an object file called `hello.o`. In order to link this object file and create the executable, you can do the linking yourself by again using the `gcc` compiler, but this time you provide the object files as the command line arguments rather than the source codes. Then you would type

```
$ gcc hello.o -o hello
```

When `gcc` sees object files, it invokes the linker automatically and links the necessary files to create the `hello` executable.

## Libraries and linking in Linux

The beauty of the C programming language is that C itself is a relatively simple and small compiler. The power of C becomes obvious when you use the wealth of precompiled libraries that have already been written that you can use in your codes. We will illustrate the use of one of the standard C libraries in an example.

Let's say we would like to use the math library in order to take the square root of the number 7 in our C code. All of the standard libraries in Linux are located in the `/usr/lib` directory. With every standard library, there is a header file that contains information about how to execute each of the functions in that library. In order to determine which header file contains the function we are looking for, we use the `man 3` command at the command prompt. Let's say we want to know which header file contains the `sqrt` function and how to use this function. To do so, we type

```
$ man 3 sqrt
```

This give us information about the function and the appropriate header file to use

```
NAME
       sqrt - square root function
```

```
SYNOPSIS
       #include <math.h>

       double sqrt(double x);
```

From this we know that in order to use the `sqrt` function, we need to use it as `sqrt(7)`, since the number 7 will be type cast as a double precision number. This also tells us that details about the `sqrt` function can be found in `/usr/include/math.h`. Our simple C program that computes square root of 7 would then be given by (Leaving out the lengthy comments for space-conservation, but you should **never** leave out comments! In this case we use shorter comments with the `//`):

```
// Program math.c
int main(void) {
  sqrt(7);
}
```

If we try and compile our program, we will see that we get an error:

```
$ gcc math.c -o math
/tmp/cciMjg1a.o: In function ‘main’:
/tmp/cciMjg1a.o(.text+0x1b): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

This is an error given to us by the linker. In this case the linker is complaining to us of an “undefined reference to sqrt”. When you get this error, it means that you did not provide enough information to the linker because you need to tell the linker where the `sqrt` function exists. You can tell the linker where certain libraries exist that must be linked with your program with the `-l` flag. In our case, the standard math library is linked with our program using the `-lm` flag, as in

```
$ gcc math.c -lm -o math
```

Compilation occurs without errors because the linker was able to find the `sqrt` function in the standard math library. Note that in our first simple program `hello.c`, we did not need to do any linking because functions in the standard C library like `printf` are automatically linked with the linker. We could, however, have use the linker implicitly with

```
$ gcc -o hello.c -lc -o hello
```

and this would have linked our program with the standard C library. But the `-lc` flag is redundant here because `gcc` does this linking automatically.

## When you need header files

Now let’s assume that in our program we would also like to use a variable that is defined in the standard math library. Variables are defined in C programs using the `#define` precompiler directive. We will get into this in more detail later, but for now let’s say we need to use the `M_PI` variable that is defined in the math library, which is in `/usr/include/math.h`. We can find the line with the `grep` command with

```
$ grep M_PI /usr/include/math.h
# define M_PI           3.14159265358979323846  /* pi */
```

which shows us that it is defined in `/usr/include/math.h` as the value of $\pi$ to 20 decimal places. If we tried to use the value of `M_PI` in our simple C program, as in

```
// Program math.c
int main(void) {
  sqrt(7*M_PI);
}
```

we get the following compiler error:

```
$ gcc -c math.c
math.c: In function ‘main’:
math.c:2: ‘M_PI’ undeclared (first use in this function)
```

It is important to understand that this error is produced by the compiler and **not** by the linker. The compiler is attempting to create the object file `math.o` but can't in this case because there is no definition of the variable `M_PI`. In order for the compiler to know what the value of `M_PI` is, we need to include the header file `math.h`. If we do so, and our code looks like

```
// Program math.c
#include <math.h>
int main(void) {
  sqrt(7*M_PI);
}
```

then we will get no errors when we try to create the object file with

```
$ gcc -c math.c
```

But remember, even by including the header file we still need to tell the linker about the library that contains the `sqrt` function if we want to create an executable.