Lab Assignment #2: The `Make` Utility

1. Objectives

The objective of this assignment is to: (i) familiarize yourself with working with multiple source files to create a single executable program; and, (ii) introduce you to the use of the `Make` utility.

2. Problem Statement

In this lab, you will examine the steps necessary to compile C++ programs from multiple source files, and practice using the `Make` utility to help automate that task.

3. Background

In the past, your software assignments have focussed on writing programs that fit within a single source file. In practice, though, most programs benefit from being spilt into multiple source files. Dividing your code across multiple files: (i) leads to better code organization and code reuse; (ii) allows a team of developers to work on the same piece of software; and, (iii) speeds up compilation time (more on that later).

Compiling from a Single File

Up until now, we've normally used a single command to compile our code and produce our program. For example:

```
g++ -g lab1.cpp -o lab1
```

It is important to know that this single command is actually performing two different tasks for us:

1. The compiler's first task is to take our program code and compile it into an "object file"[1].
2. As a second step it then "***links***" that object file with the C++ libraries[2] in order to produce our final program.

These two steps are shown in Figure 1, below:

[1] Object files do not contain C or C++ code, but contain a copy of your program that has been re-written into a form of pseudo-code that is similar to the CPU's assembly code.

[2] The C++ libraries provide us with the built-in C and C++ functions that we're familiar with, like `printf` and `cout`.

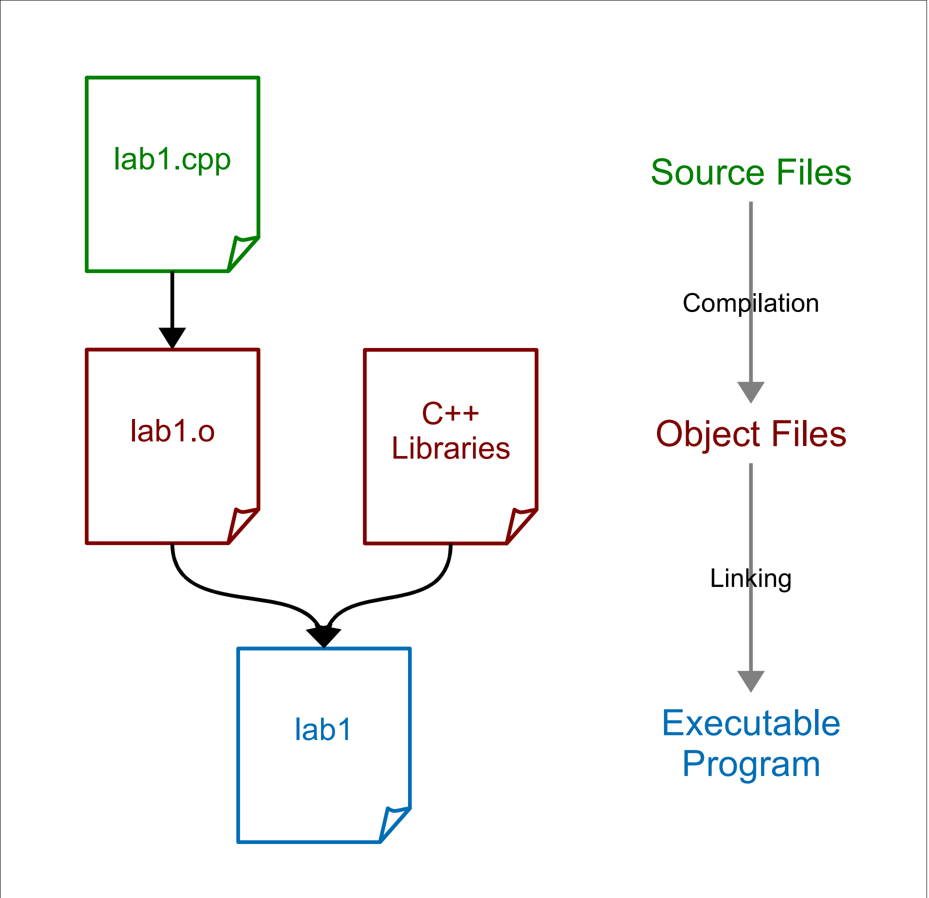

Figure 1: Basic compilation steps for a program with one source file.

In order to produce a program from multiple C++ source files we need to perform each of these steps separately. We need to: (i) compile each ".cpp" file into its own object file; and then, (ii) link these object files together in order to make our program.

Producing and Linking Object Files

If we assume that our program is comprised of two C++ source files, called "`part1.cpp`" and "`part2.cpp`", then our source tree looks like the one shown in Figure 2.

The command for converting a file containing C++ source code into an object file is:

```
g++ -g -c source file
```

and the command for linking a series of object files together in order to make a program is:

```
g++ object file 1 object file 2 [...] -o program name
```

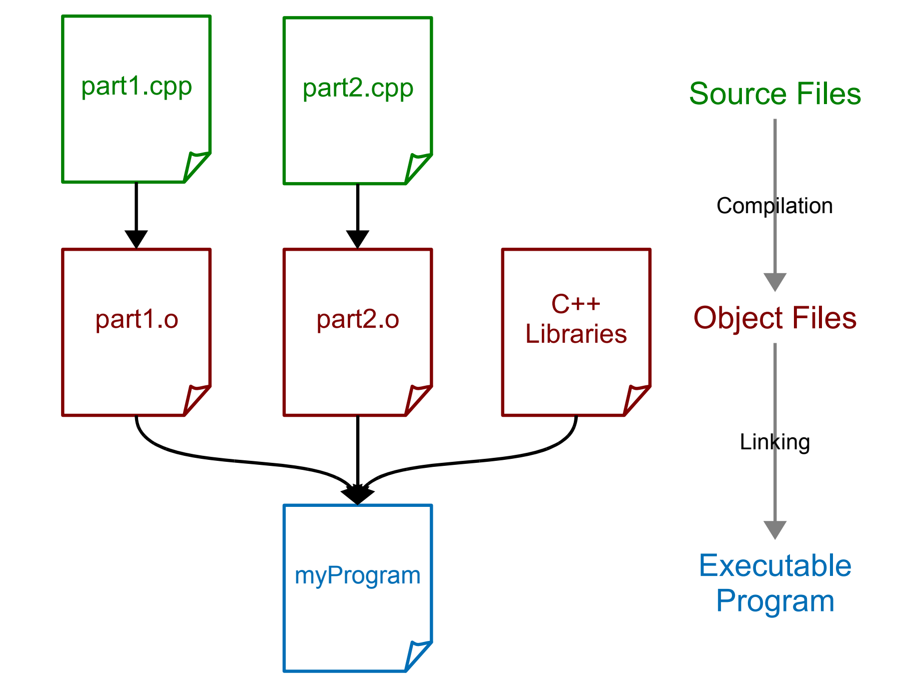

Figure 2: Compilation steps for a program with two source files.

Continuing with the example, the commands we would need to execute in order to create the program shown in Figure 2 are:

```
g++ -g -c part1.cpp
g++ -g -c part2.cpp
g++ part1.o part2.o -o myProgram
```

`g++ -g -c part1.cpp`	*(Convert part1.cpp into part1.o)*
`g++ -g -c part2.cpp`	*(Convert part2.cpp into part2.o)*
`g++ part1.o part2.o -o myProgram`	*(Link the object files to create myProgram)*

Figure 3 shows these commands, along the step that each one represents:

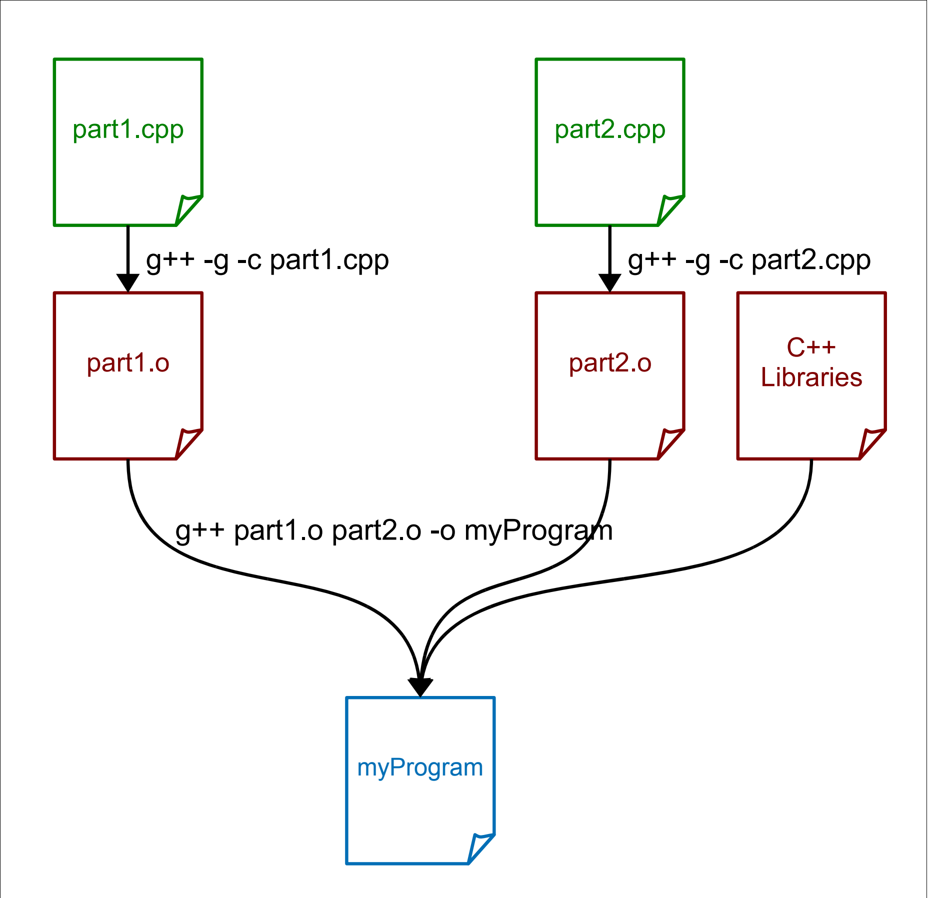

Figure 3: Commands associated with each compiler step.

The `Make` Utility

Large software projects are made up of hundreds or, in some cases, many thousands of source files. As a result, these programs require hundreds or thousands of commands to be executed in order to compile. It would take days to simply compile your program, if you had to type these commands by hand. Fortunately, there are a number of tools that can automate this task; the most popular in the Linux/Unix world is `Make`.

The basic principle behind `Make` is that you provide it with a list of files that it should "watch" (by specifying the dependencies that exist between the files), and a list of commands to run when those files are changed. You save this information in a file called `Makefile`.

The contents of your `Makefile` consists of a set of ***dependency rules*** separated by an empty line. Each dependency rule consists of a ***target*** and a ***list of dependencies*** of the target, separated by spaces, followed by zero or more (tab-indented) commands to create the target, each on a separate line:

```
target: list of dependencies
     command 1
     command 2
     command 3
     …
```

Continuing with our previous example, we can create a list of three dependency rules, starting from the bottom of our source tree (Figure 3) and working back towards the C++ source files at the top:

1. `myProgram` should be re-created if either `part1.o` or `part2.o` change.
2. `part1.o` should be re-created if `part1.cpp` changes.
3. `part2.o` should be re-created if `part2.cpp` changes.

These rules form the basis of the information that we provide to `Make`. The rules should be saved into a file called `Makefile`, in the same directory as your source files:

```
myProgram: part1.o part2.o
     g++ part1.o part2.o -o myProgram

part1.o: part1.cpp
     g++ -g -c part1.cpp

part2.o: part2.cpp
     g++ -g -c part2.cpp
```

Please note: lines with commands (*i.e.*, "g++", etc.) must start with a [Tab] character.

In the example above, the target "`myProgram`" depends on "`part1.o`" and "`part2.o`". If either `part1.o` or `part2.o` is changed, then the target `myProgram` is recreated by executing the command:

```
g++ part1.o part2.o -o myProgram
```

Similarly, the target "`part1.o`" depends on "`part1.cpp`", and if `part1.cpp` is changed, then `part1.o` is recreated by executing the command:

```
g++ -g -c part1.cpp
```

Running the "`make`" command with no arguments will execute the necessary commands, in the correct order, needed to make the first target if any of its dependencies have changed; if make is run on the above Makefile for the first time (so there are no object files yet), it will, in this case, compile and link our program:

```
[jskule@p15 ~/ece244/lab2]$ make
g++ -c part1.cpp
g++ -c part2.cpp
g++ part1.o part2.o -o myProgram
[jskule@p15 ~/ece244/lab2]$
```

Make is smart enough to tell which files have changed, and will only compile those that require it. This can save hours of compilation time on large, commerical-scale software projects:

```
[jskule@p15 ~/ece244/lab2]$ edit part1.cpp
...
[jskule@p15 ~/ece244/lab2]$ make
g++ -c part1.cpp
g++ part1.o part2.o -o myProgram
[jskule@p15 ~/ece244/lab2]$
```

4. Preparation

1. Create a new "`lab2`" directory, under the `ece244` directory in your account. Please ensure that neither of those directories can be read by other users[3].
2. Download the `main.cpp` and `main.h` files from the Lab 2 section of the course website. Please do not change these files: your program code must function with them, and they must be submitted exactly as downloaded.
3. Download the sample `Makefile` from the Lab 2 section of the course website. You can use this as the basis for the `Makefile` that you create in this assignment: please make whatever changes you wish to this file.

5. Procedure

This lab requires you to write both a short segment of C++ code, and a Makefile that can be used to compile your program.

Step 1: C++ Code

Examine the code contained in `main.cpp`. This code makes use of two functions that you must write:

```
void printMultiple (char c, int count)
void printNewline ()
```

The `printMultiple` function simply prints *`count`* copies of the character contained in *`c`*. The `printNewline` function prints a "newline" character (`endl`). In both cases, please use `cout` for the printing, and not `printf`.

[3] Please review the instructions in Lab #1 for the "`chmod`" command for more information on this.

As an example, the following code:

```
printMultiple ('A', 5);
printMultiple ('B', 4);
printNewline ();
printMultiple ('C', 3);
```

should produce the following output on the screen:

```
AAAAABBBB
CCC
```

You must create two files, in order to provide both the function declarations and the actual program code for these two functions:

- **`printFunctions.cpp`**: Your definitions (C++ code) for the `printMultiple` and `printNewline` functions.
- **`printFunctions.h`**: Your declarations (function prototypes) for the functions in `printFunctions.cpp`

Step 2: The Makefile

Your goal is to write a `Makefile`, based on the sample one you have been provided. When you run "`make`" (without any other parameters) it should produce a program, called "`lab2`", created from the `main.cpp` which you downloaded and the `printFunctions.cpp` that you created in the previous step:

```
[jskule@p15 ~/ece244/lab2]$ make
g++ -c main.cpp
g++ -c printFunctions.cpp
g++ main.o printFunctions.o -o lab2
[jskule@p15 ~/ece244/lab2]$
```

Our source tree for this lab is shown in Figure 4:

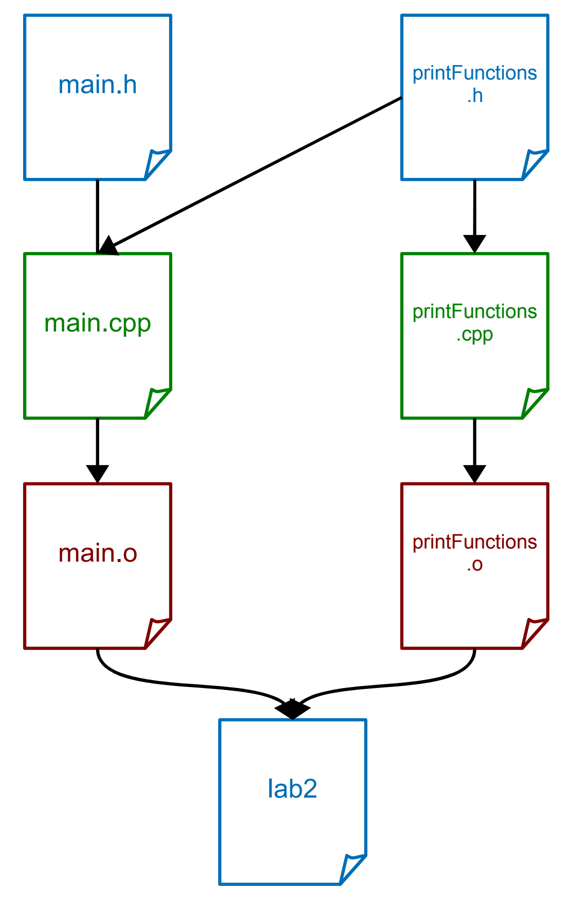

Figure 4: Source tree for "lab2". (Note that the C++ Libraries have been omitted from this diagram, for clarity.)

In addition, your `Makefile` should be expanded to run the "submitece244f" command for you when you type "make submit":

```
[jskule@p15 ~/ece244/lab2]$ make submit
submitece244f 2 Makefile main.cpp main.h printFunctions.cpp printFunctions.h
[jskule@p15 ~/ece244/lab2]$
```

The sample Makefile you are given already has two other commands defined: you can use those as a template for your "make submit".

6. Deliverables

Your submission for this lab assignment should include the following five files: Makefile, main.cpp, main.h, printFunctions.cpp and printFunctions.h. If your Makefile is complete, you should be able to type:

```
make submit
```

Otherwise, please submit your work using the usual "submitece244f" command:

```
submitece244f 2 Makefile main.cpp main.h printFunctions.cpp printFunctions.h
```