CS 15: make and diff Lab

Part I: make

Introduction

Professor Evil is hijacking a class and wants your help to fail all the students! That way, their class will look much better in comparison. They are so enthusiastic that they have done all the coding already. You only need to make a `Makefile` for the evil plan to succeed!

Review: Phases of Program Translation

You learned in CS 11 that `C++` programs cannot run in their text form. You need to translate the `C++` source code, written in plain text, into the binary representation of the particular computer's instructions.

This translation takes place in a series of stages. Once you start to write programs that have multiple modules (components/classes, stored in different files in `C++`) you must be aware of two phases: compilation and linking.

Compilation

Compilation translates a single source module/file into a binary form, called a relocatable object code file. These are stored in .o (object) files. Object files contain:

exports

exports are things that other modules can use. This might be a function like `CharLinkedList::first`.

imports

imports are things that are used by a module, but not defined in that module's `.o`, and must be supplied by some other module. If your module calls the `exit` function, for instance, then it would need to import a definition for it.

Compilation always happens on a single file. If you type

```
clang++ -Wall -Wextra file1.cpp file2.cpp
```

`clang++` will compile each file separately into a `.o` file and then proceed to the next step (linking). In this case, it stores the `.o` files in a temporary directory somewhere else, but just know they are created.

You can tell `clang++` to stop after the compilation phase by using the `-c` command line argument:

```
clang++ -c -Wall -Wextra file1.cpp file2.cpp
```

Linking

Linking combines a collection of object code (`.o`) files and the system libraries (which define things like the `<<` operator and various pre-defined functions like `exit` and `sqrt`) to form an executable binary program. In order to do this:

- All the imports of all the `.o` files have to be supplied a definition either from one of the `.o` files or from the system libraries.
- There must be exactly one `main` function.
- There cannot be more than one definition for any function (with the same number and type of parameters).

Compilation Errors and Linking Errors

Why does this matter? Because you will get error messages, and if you don't understand what translation phase they come from, you can waste **many hours of your time** looking for the solution in the wrong place! You probably already have done this!

Compilation Errors

Compilation errors include:

- syntax errors
- missing `.h` files (actually that happens before compilation, but good enough for now)
- type errors (e. g., you're calling a function with certain parameters, but they are not the types listed in the function header)
- undefined variables
- "control reaches end of non-void function"
- (and many more!)

.h files in CS 15 usually do **not** contain the function definitions, only the headers. This is enough for compilation to proceed, because it can check the types and then just list the function as an import in the `.o` file.

Linking Errors

There are far fewer linking errors:

- *Undefined reference* to a function means that a function is used somewhere (imported), but no .o file defines it. Either you've left off the relevant object code file or you omitted the function from the .o file it belongs in, or the definition differs from the promised type that the other files were compiled against.
- A special case of the above: *missing `main`*. You need a `main` function in any program. One common reason for this error is that you were trying to compile your code, but neglected to include the `-c` parameter to `clang++`. If you were trying to link your code, you may have omitted the `.o` file containing `main` (or you defined a different `main` function with non-standard parameters).
- *Multiple definitions* of the same function. One reason students sometimes run into this is that they `#include`d a `.cpp` file, which has the effect of pasting all that code into every file that includes it and thus putting the definitions in every corresponding `.o` file. Therefore, if you have multiple functions named the same thing, the linker can't decide which definition to use.

Separate Compilation and Linking

In our `Makefile`s, we explicitly separate these two phases. A `make` target for a program should do compilation **or** linking, but not both in one step. This allows us to, for example, provide different command line arguments for the compilation and linking steps. As you write larger programs, this will become necessary. It also allows us to take advantage of more sophisticated `make` techniques (that you don't have to know for today): default and pattern rules.

Example

The following commands:

```
clang++ -Wall -Wextra -c main.cpp
clang++ -Wall -Wextra -c ArrayList.cpp
clang++ -g3 -o testArrayList main.o ArrayList.o
```

can be separated in the Makefile as follows:

```
CXX = clang++
CXXFLAGS = -Wall -Wextra
LDFLAGS = -g3

testArrayList: main.o ArrayList.o
	${CXX} ${LDFLAGS} -o testArrayList main.o ArrayList.o

main.o: main.cpp ArrayList.h
	${CXX} ${CXXFLAGS} -c main.cpp

ArrayList.o: ArrayList.cpp ArrayList.h
	${CXX} ${CXXFLAGS} -c ArrayList.cpp
```

See the provided `Makefile` for details on the specifics.

Assignment

Your assignment is to make a `Makefile` that can compile and link the evil program given in the starter code. Copy the files from `/comp/15/files/lab_make/` to a `lab3` folder in your `CS15` directory. The program driver code is in `evil.cpp`. If you take a quick look at the code you will see that it uses an object of class `Roster`, whose interface is in `Roster.h` and whose implementation is in `Roster.cpp`. The class contains a simple `ArrayList` of `Student`s (a struct defined in `Student.h`). Thus, in terms of dependencies, evil depends on `Roster` which itself depends on `Student`. How would this look in the `Makefile`?

We provided a `Makefile` that does not work but has lots of documentation. In fact, it contains a whole tutorial on `Makefile`s! Working with it, follow these steps:

1. Based on your review of this document and the `Makefile`, fill in the blanks. When properly set, you should be able to build the evil program using the `make` command in your terminal. Execute it once on the file `gradebook.txt`.
2. Type `make` a second time. Why is it not compiling anything?
3. Type `ls -l` and note the time on `evil.cpp`
4. Type `touch evil.cpp` in the console. This changes the modify time as if you had edited `evil.cpp`. Try `ls -l` again and see.
5. What will happen if we type make now? Will nothing happen again? Something else? Think in advance and discuss with your neighbor.
6. Try `make`. Were you right? Discuss what happened and why with your neighbor.
7. What would happen if we touch Student.h instead? Again, verify your answer.

Part II: diff

Professor Evil has found an old working executable of the `evil` program. They provided this executable program with your starter files—it is named `evil_reference`. Professor Evil has requested that you make your version of the program *identical* to the old version. Specifically, they want to make sure that all printed outputs from your version match the outputs from the old version exactly.

You can do this using `diff`, a command-line tool that allows you to compare the contents of two different files. All differences between the two files will be reported to you by `diff`—if there are no differences, then nothing gets reported.

Redirecting Output

Because `diff` compares files, if you want to compare your program's terminal output with that of the reference, you first must save each program's terminal output in its own file. We call this *redirecting* the output to a file.

To do this, you'll first want to switch from using `tcsh` to `bash` in terminal. Both `tcsh` and `bash` are essentially programming languages—they are the languages of commands that you enter in terminal. By default, the Tufts `homework` server uses `tcsh`. While it is possible to redirect output using `tcsh`, it is a bit more complicated. Instead, you can use bash by simply typing "`bash`" on the command line and hitting enter. This will change the prompt you see on the command line, like this:

```
vm-hw03{milod}101: bash
bash-4.2$
```

Later, if you want to stop using `bash`, you can simply type "`exit`".

Now, if you want to redirect **all** terminal output (both standard output and error messages) to a file, you can do so by running your program the normal way, along with the `&>` redirection operator. For example, try running your program and redirecting the output to a file named `my.output`:

```
./evil gradebook.txt &> my.output
```

You'll notice that nothing prints to terminal this time. Instead, a file named `my.output` should have been created. Open up this file—it should look identical to the terminal output you saw before. Now, create a similar file from the reference implementation:

```
./evil_reference gradebook.txt &> ref.output
```

You should now have two separate files, `my.output` and `ref.output`, which respectively contain the output from your program and the reference implementation.

Running diff

Let's use `diff` to compare these two files. On the command line, run:

```
 diff my.output ref.output
```

If the outputs were identical (which is our goal), then `diff` wouldn't print anything, because there are no differences. However, in this case you should receive a `diff` output that looks as follows:

```
1c1
< Here is a log of failed students!
---
> Here is a log of failed students:
```

Let's break this down. First, we see `1c1`. The two numbers are line numbers. The first number is a line number from the first file given to `diff` (in this case, `my.output`), and the second number is a line number from the second file (`ref.output`). The "`c`" in the middle means that we must "change" line 1 from the first file to make it match line 1 from the second file (other options are "`a`" for add, and "`d`" for delete).

Next, we see the actual lines themselves. First, we see the line from the first file:

```
    < Here is a log of failed students!
```

followed by a `---`, and finally followed by the line from the second file:

```
    > Here is a log of failed students:
```

Can you spot the difference between the two lines? Once you see it, go in and edit your program so that it matches the reference, then re-compile it. Once again run your program and redirect its output to a file, and use `diff` to compare that file to `ref.output`. Does `diff` have any output? If not, congratulations! You have successfully matched the reference program's output. If there is a `diff` output, keep on debugging until there is no longer a difference between programs.

Using `diff` is a **crucial** skill in CS15. It's exactly how our autograder for every assignment works: we use `diff` to compare both your terminal output and any file outputs to those from a reference implementation. If you want to get a good autograder score, it is *necessary* that you use `diff` to test your program against the reference implementation we provide you.

Separately Redirecting stdout and stderr

Not so fast! It turns out that your `evil` program may still have differences from the reference implementation. Specifically, when we used `&>`, it redirected *all* termainl output from your program to the same file. But what if your program's outputs are actually from different places? For example, when you send output to `cout`, it goes to the *standard output stream* (abbreviated `stdout`), and when you send output to `cerr` it goes to the *standard* *error stream* (or `stderr`). But using `&>` lumps these different output streams into a single file.

Luckily, in `bash` we have a way to separate outputs from these two different streams. Namely, we can use `>` and `2>`, which respectively redirect `stdout` and `stderr`. Try doing this with your program:

```
./evil gradebook.txt > my.stdout 2> my.stderr
```

This will create two different files, `my.stdout` and `my.stderr`, which respectively contain your program's outputs to the `stdout` and `stderr` streams.

Now do the same with the reference implementation:

```
./evil_reference gradebook.txt > ref.stdout 2> ref.stderr
```

Finally, we can separately compare your program's `stdout` and `stderr` streams with those of the reference:

```
diff my.stdout ref.stdout
diff my.stderr ref.stderr
```

You'll notice that we are now getting outputs from `diff` that we did not get before! This means that, even though our program's total output matched the reference's, there are still differences in which stream (`cout` or `cerr`) that output was being sent. **Note:** this is also a subtle way to lose autograder points!

See if you can interpret the results you got from `diff` and determine what changes you need to make to your program to have it match the reference. It should be as simple as sending output to the correct stream. Once again redirect your output streams to files, compare against the reference with `diff`, and make sure the changes you made have resulted in the two programs matching.

Submitting your code

You will need to submit the following files:

```
Makefile
evil.cpp
roster.cpp
roster.h
Student.h
README
```

You must submit them using Gradescope to the assignment `lab_make`. Submit these files directly, do not try to submit them in a folder.