CS 202 Lab 1

Learning Targets

- I can create a makefile that compiles a simple application
- I can debug code using my IDE and/or the terminal output
- I can use cin and cout to perform console IO

Grading

This lab is graded on a "pass / needs work" basis. No partial credit is given. You must pass the autograder tests to earn full credit. Labs are **due** the Sunday of the week the lab was assigned. Late work can be submitted, but it is not eligible for regrade. If you submit a reasonable attempt at the lab by the due date, you can request a regrade by completing the google form linked in the syllabus. All resubmissions are due by the day before the test that covers the lab at 11:59 PM.

Purpose

The purpose of this lab is to give you experience in creating a makefile, as well as refactoring code from a single file into multiple cpp and header files. You have been provided with **original.c**, a c program. Your job is to use c++ **cout** and **cin** from the iostream library, rather than printf and scanf from c. You will also divide the code up among **myfunctions.cpp**, **main.cpp**, **myfunctions.h** to gain practice with using multiple files. You will construct a makefile to compile everything into a single executable.

Step 1

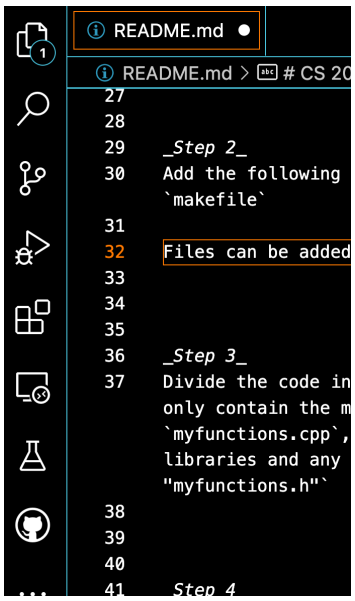
If you are on a Windows system and have VSCode with WSL installed, open the powershell by searching for it in your start menu. Right click on it, and open it in administrative mode. Navigate to the directory on your windows machine where your content for the class is, and then start VSCode by typing code . into the terminal. You can open VSCode on any of the lab machines, and if you are on Mac or Linux, you can just open VSCode by searching for it or selecting the icon.

To open a terminal in your VSCode IDE, select the terminal tab from above the title bar, then choose new terminal. This will open a terminal in the IDE. You can open a terminal anywhere in a UNIX backended system, such as Mac or Linux (not just in the IDE application). If you don't know how to access it, your operating system likely has a search for file or search for application functionality. If you are unfamiliar with navigating the file system through the use of a terminal, you can find a list of common Linux commands can be found on WebCampus under the course starting materials.

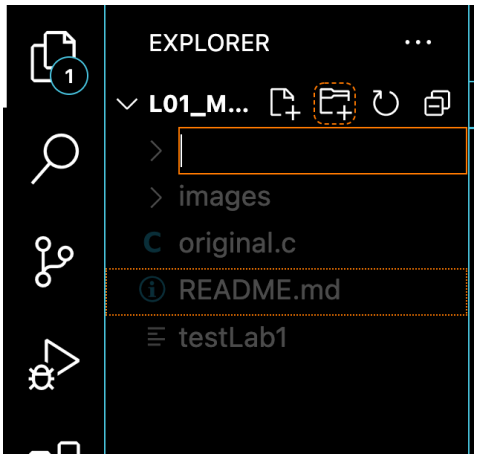
Compile the c code found in original.c using `gcc -o cCode original.c` on the commandline. This will use the gcc compiler to create an executable named cCode (the -o flag tells the compiler that the output executable name should be whatever comes immediately after, rather than the default a.out) from the original.c file. Run the code by doing ./cCode to get a feel for the expected behavior.

Step 2

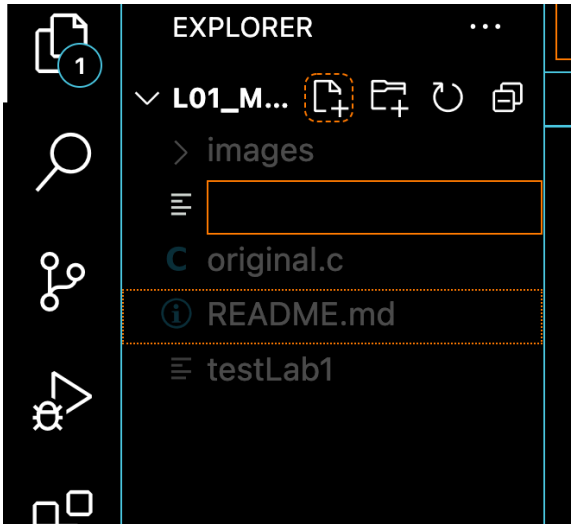
Files can be added in VSCode by clicking on the topmost icon depicting two pages in the navigation bar.



This will open an additional menu. If you move your mouse into the menu that opens, you will see a page with a plus sign, a folder with a plus sign, a circular arrow, and two squares with a minus on the foremost. If you select the folder, a box will open for you to enter the folder name.



If you select the page, a box will open for you to enter the program name with the extension (such as png, txt, c, cpp).



Clicking out of the opened menu after selecting to add a file or folder will result in the file or folder not being created.

Add the following files to your lab assignment directory: **myfunctions.cpp**, **main.cpp**, **myfunctions.h**, **makefile**

Step 3

With object oriented programming, we usually end up with a large number of files. In order to shorten our code, we typically separate declarations (remember prototypes?) into a .h file, and implementation (the actual function definitions) into a .cpp file.

Divide the code in original.c between **myfunctions.cpp**, **main.cpp**, **myfunctions.h**. The main file should only contain the main function's code. Prototypes, macros, and and included libraries should be placed in the .h file. The actual function implementation should be placed in the .cpp file.

Once you have the code divided up, include the header file in both **main.cpp** and **myfunctions.cpp**. This will allow **myfunctions.cpp** to access the declarations in **myfunctions.h** and it will allow main.cpp to access the declaration and associated definition. Include the header file in both cpp files, like so `#include "myfunctions.h"`

At this point, your code is still using the c library, stdio, which only allows printf and scanf. In c++, we will use the iostream library, which gives us access to cin, cout, getline, and a variety of other methods. Replace **#include <stdio.h>** in myfunctions.h with `#include <iostream>`. If you don't want to have to place std:: before every call to cout, cin, getline, etc, you can also add using namespace std to myfunctions.h.

Step 4

Modify the code in the three files so that it uses cout and cin, rather than scanf and printf.The general syntax for using cout is

```
cout << "Whatever you want to say" << endl; OR
cout << variable << "\n";
```

and they can be chained together like so: `cout << "I say " << variable << " " << variable << endl;`

You must manually place spaces between variables if you do not want them concatenated. Additionally, the endl function moves the output to a newline, but you can also use the \n character that you learned about in CS135 if you took it here at UNR. endl does not always need to be placed at the end of a statement- we could just as easily do this: `cout << "I say " << variable << endl << variable << endl;`

The general syntax for cin, assuming you create a variable of whatever necessary type named temp, is like so:
`cin >> temp;`

Notice that cin and cout do not care what data type the variables being used are. Cin only cares in that the data type of the variable that's being read into should match the data type of the variable you're reading in. So for example, if you needed to get an int value from the user using cin, temp would be declared as an int variable before using it with cin. Much like scanf from C, cin should not use any endlines. Just like cout, variables that need to be gathered can be cin'd by chaining. So for example, if the user was going to input firstName lastName age, our cin statement could be: `cin >> firstName >> lastName >> age;` or it could be a series of statements

```
cin >> firstName; cin >> lastName;
 cin >> age;
```

This works because the cin object that you're using stops reading as soon as it sees a space. So as long as our input from the terminal or file is always the same, we can gather input and expect cin to automatically know where one variable ends and another begins.

A note about cin: Because cin stops reading at the space character, if a user enters a string that contains spaces but we want to store that string as a single string, such as "Hello, my name is

Frank.", you will need to either user getline (more on that later in class), or you can use a variable to read in each word one at a time and then concatenate those words into a single string using the + or += operator.

Step 5

Add a file named `makefile` to your folder. Do NOT include an extension (such as .txt or .c).

Construct the makefile for myfunctions.cpp, main.cpp, myfunctions.h. It should generate **myfunctions.o**, **main.o**, and an executable named **averager** which you should run to ensure you have modified your code correctly. The purpose of a makefile is to combine all of the commandline arguments we'd have to manually run from the terminal using g++ into a single file that we can call make on. Each block of commands in the makefile is known as a recipe. The general format for each file should be:

```
executableName: main.o files1.o files2.o
	g++ -o executableName main.o files1.o files2.o

main.o: main.cpp files1.h files2.h
	g++ -c main.cpp

files1.o: files1.h files1.cpp
	g++ -c files1.cpp

files2.o: files2.h files2.cpp
	g++ -c files2.cpp

clean:
	rm *.o executableName
```

Where **executableName** is replaced with the name of the executable you want to create and run- in this case **averager**, and **files1**/**files2** can be replaced with the name of the files you're working with ignoring the extension. Typically, .h files and .cpp files with the same name are combined into a single object, so for this assignment you will have one object for main.cpp, and one object for **myfunctions.cpp** and **myfunctions.h**, along with the executable name.

The topmost line of every recipe defines the object or executable being created (.o files are object files, and they share the name of their .h/.cpp file duos with the exception of main). You should list all files that were directly included in the .h of that class. So for example, the recipe for myfunctions.o should have myfunctions.h and myfunctions.cpp after the colon, and main.o should have myfunctions.h and main.cpp on the first line (since myfunctions.h is included in main). If a .o object is created from multiple .h files, then you should list them all, but usually there's only 1 .cpp file on that top line. .h files are inheritable downstream, which means that if I

had class A include class B header file, and then I included class A's header file in main.cpp, the main.o object's topline would list the .h file for both class A and class B.

The actual compilation happens on the second, tabbed line in a recipe. We use the g++ command with the -c flag to create .o files, and the -c flag should only be called on the .cpp file in the .h/.cpp file pairing. The -o flag is used to indicate that the executable name should be whatever immediately follows, just like with c compilation.

You **MUST** use tabs on the second line of each of the recipes.

The clean command above removes the executable you create (averager) and all of the .o files using *.o.

Once your makefile is complete, generate your executable. Do this by using the make command in the terminal, within the same directory as your code. You will know if it was successful because the terminal will output each of the recipe's second lines to the terminal as they compile, and then nothing. If you check the contents of the folder you're doing work with in VSCode in the left hand menu, you should see that 2 .o objects were created, one for main and one for myfunctions, and you should see an executable named averager.

If all .o files weren't generated, or your terminal outputs an error, you have a bug. The output on the commandline will list the number of errors (sometimes a single error results in a lot of output to the terminal), and you should scroll back to the top of the output (to where you last used the make command) to see the name of the file causing the bug, followed by the line that the bug is on and a description, like so:

```
(base) sarad@Saras-MacBook-Air-2 solution % make
g++ -c driver.cpp
driver.cpp:12:10: error: invalid operands to binary expressi
std::ostream' (aka 'basic_ostream<char>') and 'SkipAdd')
    cout << newVal;
    ~~~~ ^  ~~~~~~
```

Once your executable is generated, run it. To run your executable, use ./ just like you would in c. It will look like this: **./averager.** The executable should behave in the same manner as the .c code if you have done it correctly.

Step 6

Test your code using the makefileTests executable I have provided. It can be run using `./tests` and it will tell you which test failed and which files need to be looked at and fixed. It will not tell you what needs to be fixed, so use it to locate the instructions you've been given for that file, so that you can get your code working.

Submission

Once you run the test executable and all tests are passed, remove all files from the folder that you want to submit other than the .cpp files, .h files, makefile, and makefileTest. Then, zip the file by right clicking and selecting compress to create a .zip file. Submit the zip file on WebCampus. Raise your hand for your instructor, who will come around and ask each of you to explain what the code is doing and what changes you made for your attendance grade.

Remember: the labs are step by step instructions meant to prepare you for your programming assignments. While you are allowed to resubmit labs, it's better to finish them as they are assigned so that you don't struggle with the programming assignments and tests.