CS 162 LAB #1 – Introduction & Review

Each lab will begin with a brief demonstration by the instructor or the TAs for the core concepts examined in this lab. As such, this document will not serve to tell you everything the instructor or the TAs will in the demo. It is highly encouraged that you ask questions and take notes.

In order to get credit for the lab, you need to be checked off by the end of lab. You can earn a maximum of 5 points for lab work completed outside of lab time, but you must finish the lab before the next lab. For extenuating circumstances, contact your TAs and the instructor.

This lab is worth 10 points total. Here's the breakdown:

- 1 point: Join/Sign up for Slack
- 6 points: multdiv.cpp is completed
- 2 points: code is factored into multiple implementation files and a header file
- 1 point: compilation is specified using a makefile

(1 pt) Step 1: Join/Sign up for Slack

Use this link to join/sign up for CS162 Slack channel using your OSU Email address: https://class-cs162-001-su20.slack.com/

We'll be using Slack in this course for Q&A because it's geared towards students helping other students with the class. Anyone can post or answer a question on Slack, and the instructor and TAs can revise and endorse student answers, which means you can be confident in the quality of the response.

You are *strongly encouraged* to post any class-related questions to Slack first instead of emailing the instructor or TAs. You should also get in the habit of checking in to Slack to answer other students' questions. This will not only enable everyone to get help quickly, but it will also help you improve your understanding of the material, since teaching someone else is the best way to learn something.

(6 pts) Step 2: Review: Arrays, Structs, Pointers, etc.

To review some of the material you learned in CS 161, write a program that creates a dynamic 2D array of `multdiv_entry structs` (defined at the bottom of the page). The 2D array will be used to store and print the multiplication and division tables for the values of row and column specified by the user. Note that the table will start at 1 instead of zero. This prevents us from causing a divide by zero error in the division table! Specifically, follow these steps to create the program.

1. Create a new file named `multdiv.cpp`. This is the file in which you will write your program.
2. Set your program up to read the number of rows and columns from the user as command line arguments. For example, if the user runs your program like this, you will have `row = 4, col = 5`:

```
./multdiv 4 5
```

You first need to check if the user provides correct number of arguments and exit the program with an error message if they do not.

Then, if the number of arguments is correct, continue to check if the user supplied a number before you convert the string to a number, for both `row` and `col`. **Continue to prompt for correct values, if the number isn't a valid non-zero integer**. The function `atoi()` will be helpful for this step, e.g:

```
rows=atoi(argv[1]);

cols=atoi(argv[2]);
```

3. Write a struct like this one to store a single entry in both the multiplication table and the division table.

```
struct multdiv_entry {
      int mult;
      float div;
};
```

4. Write a program that uses `row`, `col`, and your `struct` to generate and then print the multiplication and division tables. For example, if the user runs your program with these command line arguments:

```
./multdiv 5 5
```

Your program should create a 5 by 5 matrix of structs and assign the multiplication values to the `mult` variable in the struct and the division of the indices to the `div` variable in the struct. Then, print out both tables.

Your program needs to be well modularized with functions. Specifically, you should write and use the following functions:

- `bool is_valid_dimensions(char* m, char* n);`
  This function should check if the rows and cols are valid, non-zero integers. Return true if they are, false otherwise.

- `multdiv_entry** create_table(int row, int col);`
  This function should allocate space for a `row` x `col` array of your `struct` and fill it with the appropriate multiplication and division values. It should return the 2D array it creates.

- `void print_table(multdiv_entry** tables, int row, int col);`
  This function should take a 2D array as created by `create_table()` and its sizes and print out the multiplication and division tables like above.

- `void delete_table(multdiv_entry** tables, int row);`
  This function should delete all of the memory allocated to a 2D array created by `create_table()`, given the array and its size as arguments. ***This function is important.*** Your program should not have a memory leak.

At the end of the program, prompt the user if they want to see this information for a different size matrix. Make sure you do not have a memory leak.

5. Now, compile your program:

```
g++ multdiv.cpp -o multdiv
```

once your program is compiled, test it with a few different values of row and col to make sure it works.

Example Run (User inputs are highlighted):

```
./multdiv t 5

Invalid size(s)!!!

Enter an integer greater than 0 for row: 5

Enter an integer greater than 0 for col: 5

Multiplication Table:
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25

Division Table:
1.00  0.50  0.33  0.25  0.20
2.00  1.00  0.67  0.50  0.40
3.00  1.50  1.00  0.75  0.60
4.00  2.00  1.33  1.00  0.80
5.00  2.50  1.67  1.25  1.00

Would you like to see a different size matrix (0-no, 1-
yes)? 0
```

(2 pts) Step 3: Interface and Implementation files

Since we now have function prototypes and a struct that is a global user-defined type, they might be useful in other contexts. Let's factor them into a more reusable form by separating the struct and function prototypes in to a separate header file and the function definitions in to a separate implementation file. Doing this will have other benefits, such as helping us keep our code more organized and allowing us to speed up compilation (by compiling only files that have changed since the last compilation). Follow these steps:

1. Create a `multdiv.h` interface file that will contain all the function and struct declaration information we need:

```
struct multdiv_entry {
      int mult;
      float div;
};

bool is_valid_dimensions(char* , char*);
multdiv_entry** create_table(int, int);
```

```
void print_table(multdiv_entry**, int, int)
void delete_table(multdiv_entry**, int );
```

Once you've created this header file, you can remove the definition of your struct from your `.cpp` file. If you have prototypes separate from your function definitions in the `.cpp` file, you can remove those too. Replace all those by including your new header file:

```
#include "multdiv.h"
```

You should still be able to compile your program here as you did before:

```
g++ multdiv.cpp -o multdiv
```

2. Create a new implementation file named `prog.cpp`. Copy your `main()` function from your original `multdiv.cpp` file, add it to this new file, and then remove it from `multdiv.cpp`. You'll need to include your `multdiv.h` header file in the new file, too.

   Now, you'll need to compile both files together to be able to run them:

   ```
   g++ multdiv.cpp prog.cpp -o multdiv_run
   ```

   After compiling, you should be able to run your new executable:

   ```
   ./multdiv_run 5 5
   ```

(1 pt) Step 4: Create a Makefile to compile your program

As our programs (and the number of source code files we use) grow, it can become painful to continually type out long `g++` commands into the terminal each time we want to compile our code. Fortunately, there is a nice GNU/Unix utility called `make`, which allows us to write a file called a makefile to specify how to compile our code. Once we have a makefile written, compiling our code can be as simple as running the command `make`.

Let's write a makefile for our program to simplify its compilation. Follow these steps:

1. Create a new file named `Makefile`. Add the following lines to this new file (***note that the indentation must be a tab character***):

   ```
   multdiv_run:
   	g++ multdiv.cpp prog.cpp -o multdiv_run
   ```

   Now, if you type the command `make` into your terminal, you should see your code be compiled.

   We can go further, factoring our compilation into different stages and specifying dependencies for each compile step, which will allow the `make` utility to run only the

compilation steps it needs to, based on which files have changed. We can also use variables to make it easier to change some values in the makefile, such as the compiler we want to use, or the name of the executable file generated. Modify your `Makefile` to look like this:

```
CC=g++
EXE_FILE=multdiv_run

all: $(EXE_FILE)

$(EXE_FILE): multdiv.o multdiv.h prog.cpp
	$(CC) multdiv.o prog.cpp -o $(EXE_FILE)

multdiv.o: multdiv.h multdiv.cpp
	$(CC) -c multdiv.cpp
```

Now if you run `make` again, you should notice your compilation happening in stages. By default, `make` will run the first/topmost target in the `makefile`, in this case, the `all` target. To build a specific target, simply add the target name afterwards, i.e.

```
make <target>
```

We usually also add a makefile target for cleaning up our directory:

```
clean:
	rm -f *.o $(EXE_FILE)
```

You can run that makefile target by specifying it on the command line when you run `make`, i.e. `make clean`.

Show your completed work and answers to the instructor or the TAs for credit. You will not get points if you do not get checked off!

Submit your work to TEACH for our records **(Note: you will not get points if you don't get checked off with the instructor or a TA!!!)**

1. Transfer all files you've created in this lab (.h, .cpp, .txt, and makefile) from the ENGR server to your local laptop.
2. Go to TEACH.
3. In the menu on the right side, go to **Class Tools → Submit Assignment**.
4. Select **CS162 Lab1** from the list of assignments and click "**SUBMIT NOW**"
5. Select your files and click the Submit button.