Lab 1

Due No Due Date **Points** 10 **Available** after Jan 7 at 12am

Introduction & Review

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 3 points for lab work completed outside of lab time, but you must finish before the next lab. For extenuating circumstances, contact your lab TAs and the instructor.

(1 pt) Getting to Know You

As talked about in lecture, it is extremely helpful to connect with your fellow classmates for the duration of the course. The TAs will be coordinating an icebreaker activity of their choice at the beginning of this lab in support of that goal. Take the chance to interact with your TAs as well; they were in your shoes only a few terms ago and will be a valuable resource.

(6 pts) Structs, Pointers, Arrays

In this lab, you will create a dynamic two dimensional array of mult_div_values structs (defined below). The two dimensional array will be used to store the rows and columns of a multiplication and division table. Note that the table will start at 1 instead of zero, to prevent causing a divide-by-zero error in the division table!

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

The program needs to read the number of rows and columns from the user as command line arguments. You should check that the user actually supplied a number before converting the input string to an integer. Continue to prompt the user for correct values if the number isn't a valid non-zero integer. At the end of the program, prompt the user if they want to see this information for a different size matrix.

```
// One approach to handling the command line arguments...
rows=atoi(argv[1]);
cols=atoi(argv[2]);
// Now check that rows and cols are valid non-zero integers
// ...
```

For example, if you run your program with these command line arguments

```
./prog 5 5
```

Your program should create a 5 by 5 matrix of structs and assign the multiplication table to the mult variable in the struct and the division of the indices to the div variable in the struct. The mult variable is an integer, and the div variable needs to be a float (or double).

Your program needs to be well modularized with functions, including main, with 10 or less lines of code. This means you will have a function that checks if the rows are valid, non-zero integers:

```
bool is_valid_dimensions(char *m, char *n)
```

and another function that creates the matrix of structs given the m times n dimensions:

```
mult_div_values** create_table(int m, int n)
```

In addition, you need to have functions that set the multiplication and division values, as well as delete your matrix from the heap:

```
void set_mult_values(mult_div_values **table, int m, int n)
void set_div_values(mult_div_values **table, int m, int n)
void delete_table(mult_div_values **table, int m)
```

Then create functions to print the tables. After your code is functioning, test it using the following approach.

Example Run:

```
./prog 5 t
You did not input a valid column.
Please enter an integer greater than 0 for a column: 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
```

(3 pts) Separation of Files & Makefiles

Since we now have function prototypes and a struct that is a global user-defined type, then we might want to begin making an interface file that holds all of this information for us.

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

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

bool is_valid_dimensions(char *, char *);
mult_div_values** create_table(int, int);
void set_mult_values(mult_div_values **, int, int);
void set_div_values(mult_div_values **, int, int);
void delete_table(mult_div_values **, int);
```

After creating this mult_div.h file you need to #include it in your implementation file (the one with a .cpp extension). You also need to remove the prototypes and struct definition from your implementation file (e.g. delete the original prototype/struct definition lines from the .cpp file).

```
#include “./mult_div.h”
```

Now compile your program normally:

```
g++ mult_div.cpp –o mult_div
```

Let's take this a step further, and keep only your function definitions in the implementation file (mult_div.cpp), and put your main function in a separate file called "prog.cpp" Your prog.cpp file will have to #include the mult_div.h file as well. To put these files together, they need to be compiled together:

```
 g++ mult_div.cpp prog.cpp –o mult_div
```

What if we had 1,000 implementation (.cpp) files? Manually compiling all of them together would take forever and have a high chance for error. Luckily, UNIX/Linux has a built in script that makes compiling multiple files together easy called a Makefile. Just type

```
vim Makefile
```

on the command line to create it. Now modify the file following the pattern shown below:

```
<target>:
     <compiler> <file1.cpp> <file2.cpp> -o <target>
```

Note that the leading whitespace MUST be a tab (you can't just use spaces). An example of what this could look like for this lab would be:

```
mult_div:
     g++ mult_div.cpp prog.cpp -o mult_div
```

Save and exit the file. You can type "make" in the terminal to run it.

One of the other benefits of makefiles is that you can add variables to it and make compiling happen in different stages by stopping g++ after compiling and before running it through the linker. This creates object files (.o files), which you can link together

```
CC = g++
exe_file = mult_div
$(exe_file): mult_div.o prog.o
     $(CC) mult_div.o prog.o -o $(exe_file)
mult_div.o: mult_div.cpp
     $(CC) -c mult_div.cpp
prog.o: prog.cpp
     $(CC) -c prog.cpp
```

Try to make your program again. Notice all the stages. In addition, we usually add a target for cleaning up our directory:

```
clean:
     rm –f *.out *.o $(exe_file)
```

Now we can run the specific target by typing "make <target>.

```
make clean
```

Makefiles are a useful way to automate and control the program building process as your projects grow in size.

Remember, you will not receive lab credit if you do not get checked off by the TA before leaving each lab. Once you have a zero on a lab, then it cannot be changed because we have no way of knowing if you were there or not!