LAB #1 – Introduction/Logon

(2 pts) Answer icebreaker and function questions

Icebreaker**:** Let's get to know our peers. Get into small groups of 4-6, and answer the following about everyone in the group:

- What were your strengths and weaknesses in CS 161? Are you CS, ECE, or other?
- What did you like (or dislike) the most about your Spring Break 2017?
- "Design First" is one of our class themes this quarter. Discuss how designing first has helped you write the code easier. If this isn't the case for you, discuss why you don't think you should design first.

Each lab will begin with a brief demonstration by the TAs for the core concepts examined in this lab. As such, this document will not serve to tell you everything 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 3 points for lab work completed outside of lab time, but you must finish the lab before the next lab. For extenuating circumstances, contact your lab TAs and Jennifer Parham-Mocello.

Pair Programming
In this lab, you can choose a partner for pair programming. **You must be checked off together. You only need one computer for pair programming.** One person will be the driver, who controls the computer and codes, while the other is the navigator, who observes and advises. After 20 minutes, your TA will switch driver and navigator, continuing this pattern until the task is complete. Please read more about pair programming and the benefits: Pair Programming Student Handout

(5 pts) Review: Arrays, Structs, Pointers, etc.

In this lab, you will create a dynamic two dimensional array of structs that contain the **multiplication and division table of the rows and columns, starting at 1 instead of zero**. This prevents us from causing a divide by zero error in the division table!

The program needs to read the number of rows and columns from the user as command line arguments. You do need to check if the user supplied a number before you convert the string to a number. **Continue to prompt for correct values, if the number isn't a valid, non-zero integer.** At the end of the program, prompt the user if he/she wants to see this information for a different size matrix. Make sure you do not have a memory leak!!!!

```
//Checked that rows and cols are valid, non-zero integers
rows=atoi(argv[1]); cols=atoi(argv[2]);
```

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).

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

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 and cols are valid, non-zero integers, **bool is_valid_diminsions(char *m, char *n)**, and another function that creates the matrix of structs given the m x 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, call functions to print the tables. **Example:** ./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  .025  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) Interface, Implementation, and Makefile

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 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_diminsions(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 file, then you can include it into your implementation, .cpp file, and remove these prototypes and struct definition from your 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 this implementation file, i.e. **mult_div.cpp**, and put your main function in a separate implementation file called **prog.cpp**. Your prog.cpp file will have to include the mult_div.h file too. But, now how do we put these two files together? We have to compile them together, i.e. **g++ mult_div.cpp prog.cpp –o mult_div**

What if we had 1,000 implementation (.cpp) files? We do not want to do this manually anymore, right? There is a built-in UNIX/Linux script that makes this easy! This is called a Makefile. Just vim Makefile to create it. Now, add the following to the file with the spacing being a tab!!!:

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

Example:

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

Now, save and exit the file. You can type **make** in the terminal to now run this file.

Let's learn a little more… We can **add variables to the file** and make **compiling happen in different stages** by stopping g++ after compiling and before running it through the linker. This creates object files, i.e. .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>”, i.e. **make clean.**

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