LAB #10
Last Lab: Structs

Remember, if you need to get Lab #9 graded, you need to show your lab **to the TAs within 10 minutes of getting to lab**, and 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!!!

This Lab: Since this is your last lab, please make sure all your grades are in Blackboard, and if you have any disputes, then talk to your lab TAs in this lab because you will not be allowed to go back and dispute things after this lab is over!!!!

(10 pts) Practice Designing/Understanding as a Group

We need to get into a bigger group before you begin your paired-programming. Get into groups of 10 for a 30 person lab, i.e. 5 (or less) pairs in 3 different groups. Each group will have a dedicated TA as a project leader. The role of the TA is to make sure all members in the group are participating and that everyone understands the requirements of the problem being solved. In addition, your group might want to dedicate someone with good handwriting to capture your thoughts and design. Each group will begin by writing a flowchart for the solution, and then, write the pseudocode.

Since Assignment #6 is the last assignment, let's do something with it in the lab. We now have function prototypes and a struct that are global, which means we might want to begin making an interface file that holds all this information for us. As a group, think about a **netflix.h** interface file that will contain all the function prototypes and struct information you need. At this time, as a team, create this **netflix.h** file that will have a **netflix struct** with just the Netflix title and rating, i.e. **string name; and string rating**;.

In addition, you need to have functions for getting this information into the struct and printing the information. At this time, agree on the function prototype for **set_netflix_info()** and **print_netflix_info()**. As a team, create this netflix.h file with the struct definition and the function prototypes.

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 "./netflix.h"

Now, **design and write** the **main** function, as well as the **set_netflix_info()** and **get_netflix_info().** Compile your program normally: **g++ netflix.cpp –o netflix**

Makefiles: Let's take this a step further, and keep only your **function definitions** in this implementation file, i.e. **netflix.cpp**. Put your **main function** in a separate implementation file called **prog.cpp**. Your **prog.cpp** file will have to **include the netflix.h** file too. But, now how do we put these two files together? We have to compile them together, i.e. **g++ netflix.cpp prog.cpp –o netflix**

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:

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

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++
prog_file = netflix
$(prog_file): $(prog_file).o prog.o
        $(CC) $(prog_file).o prog.o –o $(prog_file)
$(prog_file).o: $(prog_file).cpp
        $(CC) –c $(prog_file).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 $(prog_file)
```

We can run the specific target by typing make <target>, i.e. **make clean.**

Enjoy your Spring Break, and see you in CS 162 Labs!!!!

Extended Learning: Exception Handling

Let's learn about exceptions. In an OO Language, we get two ways to catch some errors. Since a string object is an empty string, then we most certainly want to check that size is greater than 0 before accessing an element. For instance, say we want to

print back the first initial of a user's name, but we forgot to prompt them☺ We do not want to print the initial if the user's name is size 0. We can do a check manually:

```
if(name.size() != 0)
  cout << "Hello " << name[0] << endl;
else
  cout << "Uh Oh, You didn't initialize me!" << endl;
```

Or we can use the at() function that does a check and returns an exception when we go outside the bounds, i.e. 0 technically doesn't exist.

```
#include <stdexcept>
…


try {
  cout << "Hello " << name.at(0) << endl;
}
catch(out_of_range &e) {
  cout << "Uh Oh, You didn't initialize me!" << endl;
}
```