1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework06, put each programming exercise into its own subdirectory of this directory, zip the entire Homework06 directory up into the file Homework06.zip, and then submit this zip file to Homework #6.

Make sure that you:

- Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
- Check the contents of the zip file before uploading it. Make sure all the files are included.
- Make sure that the file was submitted correctly to MyCLasses.

All non-templated class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. All templated class structures are to be guarded and written entirely in their (.h) file. No inline coding in the class specification. In addition you must create a make file that compiles and links the project on a Linux computer with a Debian or Debian branch flavor.

2 Programming Exercises

These exercises are a bit of a continuation and combination of several previous labs, homeworks, and class discussions. Computer Science II is a course that begins your investigation of data structures, most of which are different types of containers for other data. So making these as efficient and general as possible is one of the primary goals for this course.

1. Start with the Array2D.h and main.cpp files from your completed Lab #6. The main should, of course, be the same as the one given in the lab. Change the storage structure for the Array2D class to be a vector of vectors. Keep all the same functions and functionality as specified for the pointer version but convert them to use the new storage structure of a vector of vectors. The new class should work with the given main.cpp without any alterations and produce the same output.

   There is one minor difference that we will discuss here. In the pointer version, with the indexing operator `[]` you returned nullptr if the row index was out of bounds. We cannot do that here since you will be returning a vector and not a pointer. So with this function you will just return the indexed row without any bounds checking. This will be less "safe" but acceptable, frankly that is the way indexing works on the vector in the first place.

2. This exercise is to simply test your new templated class structure. Make a new folder for this program. Copy over either one of the Array2D.h files (either the pointer one or the vector one). The program should work with either one of these so you may want to try them both. Copy over the Die class (.h and .cpp) files from Homework #3. In the Die class, remove the constant `MIN_VALUE` from the specification and replace it with a 1 in the roll function. The reason you are doing this is because the default assignment operator will not work with a constant in the specification, i.e. `d1 = d2`. Now add overloaded == and != operators that simply return if the values of the dice are the same of not. Also overload the stream out operator << to just stream out the value of the die with no spaces before or after the value. Now the class is ready to be stored in your Array2D structure.

   Write a program that will ask the user for the number of trials they would like to do. Then ask them for the number of dice to roll for each trial. Then ask them if they wish to print out the array or not. The program should then create an Array2D with the trials number of rows and the number of dice columns. The Array2D will hold Die objects (not integers). The program will then go through the entire array and roll each die. If the user selected to print the array then the array should be printed, using only the display function from the Array2D class. The program should then go through the array and see how many of the trials had all the same roll for each of the dice. Finally have the program print out the count of the trials that had all the same roll.

   Three runs of the program are below.

```
Input the number of trials: 20
Input the number of dice for each trial: 2
Do you want to display the array (Y/N): y
  5  4
  1  6
  1  5
  5  2
  1  2
  4  1
  2  5
  2  2
  5  6
  6  4
  2  3
  6  5
  6  4
  1  5
  1  4
  5  5
  6  5
  3  6
  1  1
  1  2

There were 3 matches of equal die in the 20 trials of the simulation.
```

```
Input the number of trials: 1000000
Input the number of dice for each trial: 5
Do you want to display the array (Y/N): n
There were 746 matches of equal die in the 1000000 trials of the simulation.
```

```
Input the number of trials: 100000000
Input the number of dice for each trial: 5
Do you want to display the array (Y/N): n
There were 77283 matches of equal die in the 100000000 trials of the simulation.
```