1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework03, put each programming exercise into its own subdirectory of this directory, zip the entire Homework03 directory up into the file Homework03.zip, and then submit this zip file to Homework #3.

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.

2 Programming Exercises

These exercises are a continuation of Lab #4 on dynamically allocated two-dimensional arrays. You may want to consult Lab #4 to refresh your memory on the setup of the Array2D class structure as well as its use of dynamic memory.

All class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. No inline coding in the .h files. 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.

1. Take the Array2D class code from Lab #4 and add in the following functions. Their prototypes are given as they are to be added to the specification.

   - `void resize(int r = 3, int c = 3, int defval = 0);`

     This function resizes the array to be an $r \times c$ array. If either dimension is less than 1 reset it to 1. This function is to also take the contents of the array before this call and preserve the cell values when the array is resized. So if the array is 3×3 and you resize it to 3×2 then the values in the first two columns are to be preserved. If one or both dimensions are larger than currently in the array the default value should be populated into the new cells. The example below should make the functionality of this function clear.

   - `void display(int);`

     This is an overload of the display function that was written in the lab. It is to take in a single integer parameter and use it as the column width in the printout of the array to the console. So `A.display(4)` will give each column 4 spaces to display and the values will be right justified in the column.

For example, the following main program will give the output below.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

#include "Array2D.h"

using namespace std;

int main() {
    srand(time(0));

    Array2D A(3, 5, -1);

    A.display();
    cout << endl;

    for (int i = 0; i < A.getRows(); i++)
        for (int j = 0; j < A.getCols(); j++)
            A.set(i, j, rand() % 100);

    A.display();
    cout << endl;

    A.resize(5, 3);

    A.display();
    cout << endl;

    A.resize(5, 5);

    A.display();
    cout << endl;

    A.resize(7, 2);

    A.display();
    cout << endl;

    A.resize(6, 5, 17);

    A.display();
    cout << endl;

    A.display(4);
    cout << endl;

    return 0;
}
```

Output:

```
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1

45 73 55 22 52
82 39 18 36 36
23 5 28 68 8

45 73 55
82 39 18
23 5 28
```

```
0 0 0
0 0 0

45 73 55 0 0
82 39 18 0 0
23 5 28 0 0
0 0 0 0 0
0 0 0 0 0

45 73
82 39
23 5
0 0
0 0
0 0
0 0

45 73 17 17 17
82 39 17 17 17
23 5 17 17 17
0 0 17 17 17
0 0 17 17 17
0 0 17 17 17

  45  73  17  17  17
  82  39  17  17  17
  23   5  17  17  17
   0   0  17  17  17
   0   0  17  17  17
   0   0  17  17  17
```

2. For this exercise you will be using your updated Array2D class to do a dice rolling simulation. In a new project space, copy your Array2D class files and take the Die class files from the class examples we did, use the last versions of these that require the main to set the seed of the random number generator. You will not need to alter any of these for this exercise, you will just be creating the main program.

   Create a main that does the following in the order specified below. Although there are many ways to accomplish the following task (in fact some that do not use array storage) I want you to follow the process below. We will create the entire simulation, stored in memory, and then use the data to do some simple analysis.

   What this program is going to do is ask the user for the number of trials they would like and the number of dice rolls they want to do for each trial. Then for each trial, the program will roll two dice and take their sum, this will be entered as one roll for that trial. The program will roll the two dice again, take their sum, and this will be the second roll for that same trial. This will continue for the number of rolls per trial the user entered and then we would do the same for the next trial and so on for the total number of trials requested. So if the user input 100 trials and 3 rolls per trial the program will create a 100×3 array. The first row will represent the first trial. That row will be populated with three separate dice rolls (sum of the two die). Then row 2 represents the second trial, again three dice rolls results entered on that row. So on for the 100 trials. At this point the simulation has been done and stored in the Array2D structure. The program is then going to go through the data and determine how any trials resulted in the same roll for each dice roll.

Here is an example run/output of the program.

```
Input the number of trials: 20
Input the number of dice rolls per trial: 3
Number of trials resulting in the same roll value was 1.
```

You do not need (or want) to print out the entire data set of the simulation but just for illustration the simulation that produced the above output is listed below. Note that line 4 was all 7's, which was the 1 count displayed on the final line of the above output. No other line of the data has all the same value.

```
10   3   3
 5   7  11
 9   9   5
 7   7   7
 6   6   9
 6   4   9
 9   9   3
 7  10   7
12   7   7
10  10   7
 8   9   6
 8   7  10
 6   3   8
11   7   9
 7   7   6
 3   7   8
 8   7   3
 8   7   7
 4   8   8
 8  11   9
```

The program coding order is described below.

(a) Set the seed of the random number generator to the current clock time.

(b) Declare the following variables for the simulation. An Array2D object named Simulation that uses the default constructor. Two Die objects, again using the default constructor si that they are 6 sided. Two integers, one to store the number of trials and the other to store the number of dice rolls per trial.

(c) Ask the user fo the number of trials and the number of rolls per trial.

(d) Resize the simulation array to the correct size.

(e) Do all the trials and die rolls needed and populate the simulation array. At this point no more die rolls are to be done and all counting is done after this process is complete.

(f) Go through each trial and determine if all the rolls were equal, if so, increment a counter.

(g) Display the final result as was done in the above example.

Make sure that you test your program, use small test (while printing out the data array) to make sure the logic is correct. Then remove the array printout and test it on very large arrays, say at least 1,000,000 trials using 5–10 die rolls per trial.

3. In this exercise you will be creating card and deck class structures to simulate a standard poker deck of 52 cards. In the card class the data is to be stored as integer values for the value and the suit. The value is to be a number between 1 and 13. The Ace is 1, 2–10 are the numeric cards, 11 is a Jack, 12 a Queen, and 13 a King. The suits are numbered 1–4 in the order of D, C, H, S. The member functions are as follows. As usual, all data is private.

   - `Card()`: Default constructor will set the value and suit both to 0, that is, a card that does not exist.
   - `Card(int v, int s)`: Constructor that sets value to v and suit to s. You may default the values so that a separate default constructor is not needed.
   - `string toStringFace()`: Returns a string of the face value of the card, use A, J, Q, and K for the face cards.
   - `string toStringSuit()`: Returns a string of the suit of the card, use D, C, H, and S.
   - `bool equals(Card card2)`: Returns true is the two cards are identical, both value and suit are the same. Returns false otherwise.
   - `string toString()`: Returns a sting of the card in condensed form, for example, 2C, AS, JD, 10H. There should be no space between the value and the suit.
   - `string toString(bool space)`: Returns a sting of the card in condensed form, for example, 2C, AS, JD, 10H. If space is true then there should be one space between the value and the suit, for example, 2 C, A S, J D, 10 H. Again you may default this to do the previous function as well.
   - `bool greater(Card card2)`: Returns true if the card has a higher value than card2. The suit value does not matter.

   The Deck class is to store an array of 52 cards and an integer value for the top of the deck. This is a quick way to do dealing without changing the deck contents. When you create a deck or after you shuffle you will reset the top to 0. Here is an example of how you can use the top. Say the beginning of our shuffled deck is as follows.

   ```
   2D QS 5H 4C KD JD QH 6C 9C 9S KH 6H QD
   ```

   So 2D is at index 0, QS index 1, 5H index 2, and so on

   Since the top is 0 then deck[top] (the 2D) is the card that is next to be dealt. So when you deal a card all you need to do is return the card at the top position then increment top. Now top is 1 and the next card to deal is deck[top] (the QS), a deal will return this card and increment top to 2 and hence another deal will return 5H and increment top to 3, and so on.

   - `Deck()`: Constructor that creates the deck of cards.
   - `void PrintDeck()`: Prints out the deck in a single like using no space between the value and suit for each card and one space between the cards.

- `void ShuffleDeck()`: Shuffles the deck of cards.
- `Card dealCard()`: Deals the next card off the top of the deck.
- `Card getCard(int i)`: Gets the card at index i in the deck.
- `void reset()`: Resets the top to 0;

Once these classes are constructed test them with the following program.

```
#include <iostream>

#include "Card.h"
#include "Deck.h"

using namespace std;

int main() {
  Deck deck1, deck2;

  deck1.PrintDeck();
  deck1.ShuffleDeck();
  deck1.PrintDeck();
  cout << endl;

  for (int i = 0; i < 10; i++)
    cout << deck1.dealCard().toString() << " - ";
  cout << endl;

  deck1.reset();

  for (int i = 0; i < 5; i++)
    cout << deck1.dealCard().toString() << " - ";
  cout << endl;

  for (int i = 3; i < 10; i++)
    cout << deck1.getCard(i).toString() << " - ";
  cout << endl;

  deck2 = deck1;

  cout << endl;
  deck1.PrintDeck();
  deck2.PrintDeck();

  cout << endl;
  deck1.ShuffleDeck();
  deck1.PrintDeck();
  deck2.PrintDeck();

  return 0;
}
```

The output should look something like the following. The deck prints will be on a single line and your ordering will be different.

```
AD 2D 3D 4D 5D 6D 7D 8D 9D 10D JD QD KD AC 2C 3C 4C 5C 6C 7C 8C 9C 10C JC QC KC
AH 2H 3H 4H 5H 6H 7H 8H 9H 10H JH QH KH AS 2S 3S 4S 5S 6S 7S 8S 9S 10S JS QS KS

6C 7S 5S 3C 4H QC 7C KS 5C AS 4S 2C JC 8D 4C KH KD 9H 9D 6H 6S QD 8C 10D 5H 3D
JS JH JD 6D 2D QS 7D 4D 3S 9C AD 10S 8H 8S QH 3H KC 9S 2H AH 10H 2S 5D 10C 7H AC

6C - 7S - 5S - 3C - 4H - QC - 7C - KS - 5C - AS -
```

```
6C - 7S - 5S - 3C - 4H -
3C - 4H - QC - 7C - KS - 5C - AS -

6C 7S 5S 3C 4H QC 7C KS 5C AS 4S 2C JC 8D 4C KH KD 9H 9D 6H 6S QD 8C 10D 5H 3D JS
JH JD 6D 2D QS 7D 4D 3S 9C AD 10S 8H 8S QH 3H KC 9S 2H AH 10H 2S 5D 10C 7H AC

6C 7S 5S 3C 4H QC 7C KS 5C AS 4S 2C JC 8D 4C KH KD 9H 9D 6H 6S QD 8C 10D 5H 3D JS
JH JD 6D 2D QS 7D 4D 3S 9C AD 10S 8H 8S QH 3H KC 9S 2H AH 10H 2S 5D 10C 7H AC

KS KD 8C 2D 2C QC 9C 10H KC AH 9H 7D 10D AD 6C 7C KH JS 2S JD 4S 7H 7S 3S 3D 8S JH
6S 5H JC 8D 5S 3C QH 9D 6H 10S 5C QS 3H 4C 2H AC 4H 9S 5D 8H QD AS 10C 4D 6D

6C 7S 5S 3C 4H QC 7C KS 5C AS 4S 2C JC 8D 4C KH KD 9H 9D 6H 6S QD 8C 10D 5H 3D JS
JH JD 6D 2D QS 7D 4D 3S 9C AD 10S 8H 8S QH 3H KC 9S 2H AH 10H 2S 5D 10C 7H AC
```

Note: You can use the `random_shuffle` command from the algorithm library to shuffle the deck. If you read the documentation on this command you probably saw that it is deprecated, which means that it may not exist in future versions of C++. In general you should stay away from using deprecated functions because it reduces the lifetime of your code. The `random_shuffle` command has been replaced by the `shuffle` command. This takes a couple more lines to set up, you need to create a random device and a generator. Here is a quick way to do this.

```
std::random_device rd;
std::mt19937 g(rd());
std::shuffle(A, A + 10, g);
```

In the above example, A is an array of 10 items (data type here is not important). Also, if you included the std namespace you do not need std::. What is going on here is you first created a random device object called rd, then you created a generator that uses this device. The generator is a mt19937 (Mersenne twister engine). Finally, you shuffled the array using this random number generator. Yes there are others, and there are different distributions you can choose from. To use this you also need to include the `random` library. You can find more information on this at `https://en.cppreference.com/w/cpp/numeric/random`.

4. This exercise uses the card and deck classes you just created. Create a new directory and copy all of the card and deck files to the new directory.

   A *derangement* is when you have a list of objects and then mix them up so that no object is in the same position as it started. For example, if we have the list `1 2 3 4 5`, then the list `2 5 4 1 3` is a derangement of the original list but `2 1 3 5 4` is not since 3 is in the same position as where it started.

   The same can be done with larger lists and is a common game using a deck of cards. Take a new deck of cards out of its wrapper and then bet someone that no matter how many times they shuffle the deck there will be at least one card in the same position as it was in the originally manufactured deck. The question is, would you take that bet? Most people do since they feel that if they shuffle the deck enough times then the cards will all be mixed up.

   In the following printouts, the top row is the original order of the deck and the second row is the shuffled deck. Looking at the first run it is clear that this is not a derangement.

```
AH 2H 3H 4H 5H 6H 7H  8H 9H 10H JH QH KH AD 2D 3D 4D 5D 6D 7D 8D 9D 10D JD QD KD
AS 6S 5C 9S 4H 6H 8H 10C 2H  9D 7C AC 8S 3C 6D 4C 5H QD 2D 5S QC 3S  7H 2C AD KD


AC  2C 3C 4C 5C 6C 7C 8C 9C 10C JC QC KC AS 2S 3S  4S 5S 6S 7S 8S 9S 10S JS QS KS
JD 10S AH JH 5D 4S QH 2S 8D  JS 7S KS KC 9H 8C 3D 10D 6C KH 9C QS 4D 10H JC 7D 3H
```

On the other hand, the following is a derangement,

```
AH  2H 3H 4H 5H 6H  7H 8H 9H 10H JH QH KH  AD 2D 3D 4D 5D 6D 7D 8D 9D 10D JD QD KD
10C 5S 3C 7H KH 2S 10D 5D JC 10S AS 2H QC 10H JS 4S 7S 5H 4D KD AC 3D  KS 8D 9S 4C


AC 2C 3C 4C 5C 6C 7C 8C 9C 10C JC QC KC AS 2S 3S 4S 5S 6S 7S 8S 9S 10S JS QS KS
KC 4H 3S 9H 9D 2C 6D AD JH  8C JD 8H QD 6C 2D 6H AH 7D 3H 9C QH 6S  8S QS 5C 7C
```

Write a program that will simulate the shuffling many times and determine if the shuffle produces a derangement. We will count the number of derangements and then find the probability of a shuffled deck being a derangement. This will answer the question of whether or not you should take that bet.

The program should take the number of trials to be done from the user. For each trial, you will have two decks of cards, one you will shuffle and the other you will leave alone as the original deck order. Go through the two decks card by card, if there is a match of a pair of cards then the shuffle is not a derangement and if there are no matches then the shuffle is a derangement. Keep a count of the number of derangements found in the number of trials the user wanted. Then calculate the probability of a derangement by dividing the number of trials into the number derangements found. Also calculate one over the probability. This program is not long if you use the functions in the card and deck classes efficiently.