1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Lab04, put each programming exercise into its own subdirectory of this directory, zip the entire Lab04 directory up into the file Lab04.zip, and then submit this zip file to Lab #4.

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 Exercise

All class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. The .h file is to have no inline code. 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.

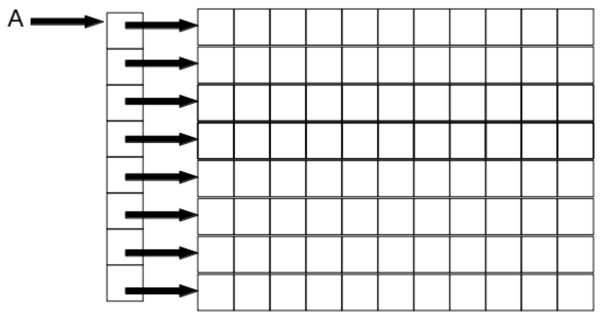
The main idea for this assignment is to get some more familiarity with class structures and object oriented programming in C++. This lab is based on extending the 2-D dynamic array to a class structure so that it is easier to use and better encapsulated. Recall, to create an 8×11 array of integers, dynamically, we would first make 8 rows of pointers.

```cpp
int** A = new int*[8];
```

Then for each row we would create an array of 11 integers.

```cpp
for (int i = 0; i < 8; i++)
    A[i] = new int[11];
```

In memory the layout of the data would look like the following.



Although there are a lot of pointers in this setup the syntax to access any position in the array is just your standard bracket notation. For example, `A[3][5] = 4;` would set the $(3, 5)$ cell to 4 and `cout << A[2][7];` would print the contents of the $(2, 7)$ cell to the console. As with any array in C++ it is possible to read and write outside the allocated memory space, which is something we do not want to do.

Freeing the memory is another aspect we need to be careful about. In the above example we crated 9 arrays, 8 arrays of integers and one array of pointers to integers. These must each be deleted separately. For our example above the following will free the allocated memory.

```
for (int i = 0; i < 8; i++)
    delete[] A[i];
delete[] A;
```

For this exercise, create a class structure named `Array2D` which has the following specification.

```
class Array2D {
private:
    int** A = nullptr;
    int rows;
    int cols;

public:
    Array2D(int r = 3, int c = 3, int defval = 0);
    ~Array2D();

    void display();
    int getRows();
    int getCols();
    void set(int, int, int);
    int get(int, int);
};
```

- The data is be in the private section, all functions are public. The data will be the array of pointers, rows that will store the number of rows the array is storing and cols will be the number of columns the array is storing.
- The constructor will take in 3 parameters, the r, c, and defval. As you can see from the specification, there are default values for each of these. r is the number of rows, c is the number of columns, and defval is the value that the initial array will be populated with. So the default constructor will create a 3×3 array populated with 0. In the constructor, make sure that the number of rows and the number of columns are not less than 1, so if r or c have values less then one, reset them to one before allocating memory.
- The destructor will free the memory allocation made for the array.
- display: Will simply print the array to the console. Don't worry about the columns lining up, just print each row on its own line with a space or two (or a tab if you would like) between each of the entries on the row.
- getRows: Will simply return the number of rows being stored in the array.

- getCols: Will simply return the number of columns being stored in the array.
- set: Will take in three parameters, the first is the row index, the second is the column index, and the third is the value to be stored in that position in the array. If the index is out of the allocated memory bounds the program should print the message `Index out of bounds.` to the console and not update the array in any way.
- get: Will take in two parameters, the first is the row index, and the second is the column index. The function will return the value in that cell of the array. If the index is out of the allocated memory bounds the program should print the message `Index out of bounds.` and just return 0.

With the above class the following main will produce 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);
    Array2D B;

    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.set(10, 3, -15);
    A.display();
    cout << endl;

    A.set(1, 3, -15);
    A.display();
    cout << endl;

    cout << A.get(2, 2) << endl;
    cout << A.get(2, 20) << endl;
    cout << endl;
    B.display();

    return 0;
}
```

Output:

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

```
21 31 3 94 50
80 43 77 4 8
41 81 10 29 8

Index out of bounds.
21 31 3 94 50
80 43 77 4 8
41 81 10 29 8

21 31 3 94 50
80 43 77 -15 8
41 81 10 29 8

10
Index out of bounds.
0

0 0 0
0 0 0
0 0 0
```

As a short note on this class structure. Even though it is a fairly simple "wrapper" class to a two-dimensional array it has the added features of,

1. Being a safe-array that will not allow array access outside the allocated space.
2. It is nicely *encapsulated* and uses *data hiding* so that anyone who is using it in their program does not need to know the underlying structure of the storage. Look over the example main program above, there is no indication that you are using pointers and dynamic memory allocation for the array.
3. Since the memory allocation is dynamic, the array is being stored in the heap and hence the person using this class structure can create much larger arrays than they would be able to using the "program memory" approach of `int A[r][c];`.
4. Since in a class structure you have the control of the constructor you can populate the array with default values on creation. Hence you do not have the problem of an array full of garbage.

One thing that would be nicer would be for the person using this class to be able to use the syntax `A[1][3] = -15;` instead of `A.set(1, 3, -15);`. We can make this happen with operator overloading, that we will cover shortly in this course.