CSCI-1200 Data Structures — Spring 2017
Homework 3 — Dynamic Matrices

In this assignment you will build a custom class named *Matrix*, which will mimic traditional matrices (the plural of matrix). You will not be expected to have intimate knowledge of matrices, but if you are curious you can read more about them online: https://en.wikipedia.org/wiki/Matrix_(mathematics)

Matrices are used in many different applications, and over the years many optimizations, tricks, and numerical methods have been developed to quickly handle matrix operations and solve more complicated problems.

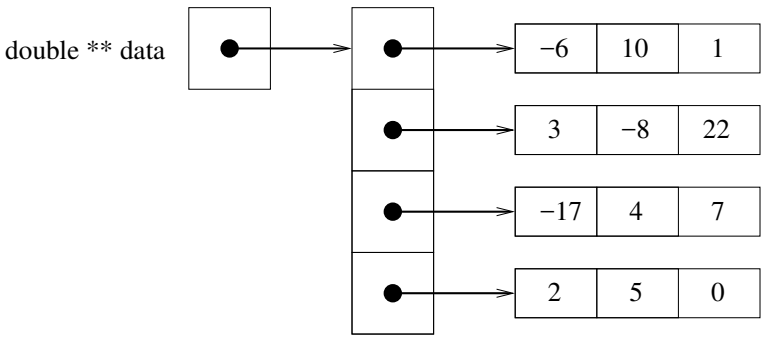
Building this data structure will give you practice with pointers, dynamic array allocation and deallocation, 2D pointers, and class design. The implementation of this data structure will involve writing one new class. You are not allowed to use any of the STL container classes in your implementation or use any additional classes or structs. You will need to make use of the `new` and `delete` keywords. Please read the entire handout before beginning your implementation. Also, be sure to review the Lecture 8 notes and our implementation of the `Vec` class mimicking STL's `vector`. You may also want to review our discussion of 2D dynamic arrays from Lecture 6.

The Data Structure

A matrix is a two-dimensional arrangement of numbers. In this assignment we will assume every matrix contains `double`s. We refer to the size of a matrix with m rows and n columns as an $m \times n$ matrix. For example, shown below is a 4×3 matrix:

$$\begin{bmatrix} -6 & 10 & 1 \\ 3 & -8 & 22 \\ -17 & 4 & 7 \\ 2 & 5 & 0 \end{bmatrix}$$

We will represent the data inside our `Matrix` class by using a two-dimensional array. Because a matrix may be any size, you will need to use dynamic memory for this task. The same matrix shown above can be represented like so:



We will denote $a_{i,j}$ as the value in matrix A that is in row i and column j. So a general matrix can be described as:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-2} & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-2} & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2,0} & a_{m-2,1} & \cdots & a_{m-2,n-2} & a_{m-2,n-1} \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-2} & a_{m-1,n-1} \end{bmatrix}$$

Basic Functionality

The `private` section of your class will be fairly small, and the main challenge will be working with the dynamic memory as you implement features to make the class functional. You can implement these methods in any order; we start by mentioning a few that will make debugging easier.

The first thing we suggest is writing a constructor that takes two `unsigned int`s: a *rows* count and a *columns count*, and a `double` *fill* value. The constructor should create a data representation of a *rows*× *columns* matrix with every value initialized to *fill*. If either dimension is 0, the resulting matrix should be of size 0×0.

Your class must support the equality operator `==` and the inequality operator `!=`. Two matrices are considered to be equal if they have the same value in every position. In other words, matrices A and B are equal if and only if $(\forall_{i,j} | i \in \{0, 1, \ldots, m-2, m-1\}, j \in \{0, 1, \ldots, n-2, n-1\})\, a_{i,j} = b_{i,j}$. $\forall$ is a common shorthand for "for all," so $\forall_{i,j}$ means "for every value of i and j." $\in$ is a common shorthand for "in".

Since a matrix has two dimensions, you will need to implement *num_rows()* and *num_cols()* which return the number of rows and the number of columns in the matrix respectively.

We may want to change a previously filled matrix to an empty one, so you must write a *clear()* method as well. This method should reset the number of rows and columns to 0, and deallocate any memory currently held by the `Matrix`.

Naturally we want to be able to access data stored in the `Matrix` class. To do so we will implement a "safe" accessor called *get()*, which takes in a row, a column, and a `double`. If the row and column are within the bounds of the matrix, then the value at $a_{row,col}$ should be stored in the `double`, and the function should return `true`. Otherwise the function should return `false`.

A complementary, but similar task to accessing data is to be able to set a value at a particular position in the matrix. This is done through a safe modifier called *set()*. This function also takes in a row, column, and a `double` value. *set()* returns false if the position is out of bounds, and true if the position is valid. If the position is valid, the function should also set $a_{row,col}$ to the value of the `double` that was passed in.

Overloaded Output Operator

At some point, it is probably a good idea to write a method to do output for us. Unlike previous classes where we wrote a method to do the printing, we will instead rely on a non-member overload of the *operator*<<. We have practiced overloading other operators for calls to `std::sort()` before, and this will be similar. Outside of the `Matrix` class definition, but still in your `.cpp` and `.h` files, you should write the following operator:

```
std::ostream& operator<< (std::ostream& out, const Matrix& m)
```

This will allow us to print one or more outputs sequentially. All of the following code should work if your *operator*<< is implemented correctly:

```
Matrix m1;
Matrix m2;
std::ofstream outfile(output_filename); //Assuming we already had the filename
std::cout << m1 << m2 << std::endl;
outfile << m1;
outfile << m2 << std::endl;
std::cout << "Done printing." << std::endl;
```

Let us assume in the above example that:

$$m1 = \begin{bmatrix} \end{bmatrix}, \quad m2 = \begin{bmatrix} 3 & 5.21 \\ -2 & 4 \\ -18 & 1 \end{bmatrix}$$

Then the output should look something like this:

```
0 x 0 matrix:
[ ]
3 x 2 matrix:
[ 3 5.21
 -2 4
 -18 1 ]

Done printing.
```

We will ignore whitespace, but we do expect that your operator outputs the elements of the matrix in the right order, that the size output comes before the matrix and follows the format shown below - one row per line, and spacing between elements. Note that even in these examples, the alignment is not ideal. We would rather you focus on the task of implementing the `Matrix` class correctly and handling memory properly instead of focusing on making the output pretty.

Simple Matrix Operations

To start with, we introduce some basic matrix operations. The first is the method *multiply_by_coefficient()*, which takes a `double` called a coefficient. The method should multiply every element in the matrix by the coefficient. For example:

$$m1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad m1.multiply\_by\_coefficient(5) \Longrightarrow \begin{bmatrix} 5 & 10 \\ 15 & 20 \end{bmatrix}$$

Another common operation is to swap two rows of a matrix. This will be accomplished by the method *swap_row()*, which takes two arguments of type `unsigned int`: a *s*ource row number and a *t*arget row number. If both rows are inside the bounds of the matrix, then the function should switch the values in the two rows and return `true`. Otherwise the function should return `false`.

For example:

$$m1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad m1.swap\_rows(1,2) \Longrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 4 & 5 & 6 \end{bmatrix}$$

NOTE: With the basic functions and swap_row() done, the tests related to the provided *rref()* function in *matrix_main.cpp* can be called. We do not explain the function in detail here, and you don't need to know how it works, but computing the Reduced Row Echelon Form (RREF) can be used to find an inverse matrix, which is important in many fields. We use a simple to implement method called Gauss-Jordan Elimination, which you can read about here: https://en.wikipedia.org/wiki/Gaussian_elimination . There are other techniques for finding the RREF that are better, but we chose this one for its simplicity.

It is common to need to "flip" a matrix, a process called transposition. You will need to write the *transpose()* method. Formally, transposition of $m \times n$ matrix A into $n \times m$ matrix A^T is defined as:
$(\forall_{i,j} | i \in \{0, 1, \ldots, m-2, m-1\}, j \in \{0, 1, \ldots, n-2, n-1\})\, a^T_{i,j} = a_{j,i}$

$$m1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad m1.transpose() \Longrightarrow \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Binary Matrix Operations

Binary matrix operations are ones that involve two matrices. To keep things simple, we will write them as methods (not operators) that are inside the class definition, so the current `Matrix` object will always be the "left-hand" matrix, A. You will be required to implement both *add()* and *subtract()*. Both functions take in just one argument, a second `Matrix` which we will refer to as B, and modify A if the dimensions of A and B match. If the dimensions match, the functions should return true, otherwise they should return false.

Addition of two matrices, $C = A + B$, and subtraction of two matrices, $D = A - B$ are formally defined as:
$(\forall_{i,j} | i \in \{0, 1, \ldots, m-2, m-1\}, j \in \{0, 1, \ldots, n-2, n-1\})\, C_{i,j} = a_{i,j} + b_{i,j}$
$(\forall_{i,j} | i \in \{0, 1, \ldots, m-2, m-1\}, j \in \{0, 1, \ldots, n-2, n-1\})\, D_{i,j} = a_{i,j} - b_{i,j}$

Consider these two matrices:

$$m1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad m2 = \begin{bmatrix} 4 & 16 & 25 \\ 14 & 3.4 & 3.64159 \end{bmatrix}$$

$$m1 + m2 = \begin{bmatrix} 1+4 & 2+16 & 3+25 \\ 4+14 & 5+3.4 & 6+3.64159 \end{bmatrix} = \begin{bmatrix} 5 & 18 & 28 \\ 18 & 8.4 & 9.64159 \end{bmatrix}$$

$$m1 - m2 = \begin{bmatrix} 1-4 & 2-16 & 3-25 \\ 4-14 & 5-3.4 & 6-3.64159 \end{bmatrix} = \begin{bmatrix} -3 & -14 & -22 \\ -10 & 1.6 & 2.35841 \end{bmatrix}$$

Harder Matrix Operations

- get_row() - returns a dynamically allocated copy of the row
- get_col() - returns a dynamically allocated copy of the column
- quarter() - "weird" matrix operation. Explain what happens when we have an odd number of columns, let them figure out the odd number of rows on their own.

If we want to get the contents of an entire row or column, it's annoying to have to extract the values one by one using *get()*, especially since our implementation is a "safe" accessor so we can't use some of the coding shortcuts we normally use. To fix this, you will implement two more accessors, *get_row()* and *get_column()*. Both functions take one `unsigned int` and return a `double*`. For *get_row()* the argument is the number of row to retrieve, while for *get_col()* the argument is the number of the column to retrieve. If the requested row/column is outside of the matrix bounds, the method should return a pointer set to `NULL`.

The final method we expect you to implement, *quarter()*, is not a traditional matrix operation. The method takes no arguments and returns a `Matrix*` containing four `Matrix` elements in order: an **U**pper **L**eft (UL) quadrant, an **U**pper **R**ight (UR) quadrant, a **L**ower **L**eft (LL) quadrant, and finally a **L**ower **R**ight (LR) quadrant. All four quadrants should be the same size. Remember that when a function ends all local variables go out of scope and are destroyed, so you will need to be particularly careful about how you construct and return the quadrants. On the next page are two examples of the quarter operation:

$$m1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \quad m2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

$$m1^{(UL)} = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad m1^{(UR)} = \begin{bmatrix} 3 & 4 \end{bmatrix} \quad m1^{(LL)} = \begin{bmatrix} 5 & 6 \end{bmatrix} \quad m1^{(LR)} = \begin{bmatrix} 7 & 8 \end{bmatrix}$$

$$m2^{(UL)} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \quad m2^{(UR)} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \quad m2^{(LL)} = \begin{bmatrix} 5 & 6 \\ 9 & 10 \end{bmatrix} \quad m2^{(LR)} = \begin{bmatrix} 7 & 8 \\ 11 & 12 \end{bmatrix}$$

Testing and Debugging

We provide a `matrix_main.cpp` file with a wide variety of tests of the `Matrix` class. Some of these tests are initially commented out. We recommend that you get your class working on the basic tests, and then uncomment the additional tests as you implement and debug the remaining functionality. Study the provided test cases to understand the arguments and return values.
Note: Do not edit the provided `matrix_main.cpp` *file, except to uncomment the provided test cases as you work through your implementation and to add your own test cases where specified.*

The `assert()` function is used throughout the test code. This is a function available in both C and C++ that will do nothing if the condition is true, and will cause an immediate crash if the condition is false. If the condition is false, your command line should show the assertion that failed immediate prior to the crash.

We recommend using a memory debugging tool to find memory errors and memory leaks. Information on installation and use of the memory debuggers "Dr. Memory" (available for Linux/MacOSX/Windows) and "Valgrind" (available for Linux/OSX) is presented on the course webpage:

http://www.cs.rpi.edu/academics/courses/spring17/ds/memory_debugging.php

The homework submission server will also run your code with Dr. Memory to search for memory problems. Your program must be memory error free and memory leak free to receive full credit.

Your Task & Provided Code

You must implement the `Matrix` class as described in this handout. Your class should be split between a `.cpp` and a `.h` file. You should also include some extra tests in the *StudentTest()* function in `matrix_main.cpp`.

When implementing the class, pay particular attention to correctly implementing the copy constructor, assignment operator, and destructor. As you implement your classes, be careful with return types, the `const` keyword, and passing by reference.

If you have correctly implemented the `Matrix` class, then running the provided `matrix_main.cpp` file with your class, should produce the output provided in `sample_output.txt`. We are not going to be particularly picky about differences in whitespace, but you should be making an effort to try and match both spacing and newlines between our output and your output.

Submission

You will need to submit your `matrix_main.cpp`,`Matrix.cpp`,`Matrix.h`, and `README.txt` file. Be aware that Submitty will be using an instructor copy of `matrix_main.cpp` for most of the tests, so you must make sure your `Matrix` implementation can compile given the provided file.

Be sure to write your own new test cases and don't forget to comment your code! Use the provided template `README.txt` file for notes you want the grader to read. Fill out the order notation section as well in the `README.txt` file. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

NOTE: If you earn enough points on Submitty by 11:59pm on Wednseday, Feb 15th, you can submit your assingment by Friday, Feb. 17th without being charged a late day. Once the Submitty submission is open, there will be a note at the top with details on how many points you need and on which test cases.

Extra Credit Opportunity

For extra credit, you may implement another method in the `Matrix` class called *resize*. This method should take in three arguments: a number of rows, number of columns, and finally a fill value. The function should always change the internal matrix to be $rows \times columns$ in size, copying over any values that were in the bounds of the original matrix. If the new matrix is larger, any positions that are in the new matrix, but not the old one, should have the fill value.

If you do this, make sure to fill out the section about extra credit in your `README.txt`, and write some tests in *ExtraCreditTest()*.