CSCE 2014 – Programming Project 3

Midpoint Due Date – 6/29/17 at 11:59 PM
Final Due Date – 7/7/17 at 11:59 PM

1. Problem Statement:

In this project, you will gain more experience with Abstract Data Types (ADTs) by implementing a **Matrix** class in C++. You will be provided with a file (`Matrix.h`) that contains the complete interface specification for that class, as well as a test suite (`main.cpp`) that tests each Matrix operation in isolation. Your job will be to provide implementations of each of the public interface methods in `Matrix.h`.

Background

In mathematics, a **matrix** is a 2D grid of numbers. For example, this is a matrix of size **3 x 3** (3 rows and 3 columns):

$$\begin{matrix} 1 & 5 & 7 \\ 2 & 4 & 0 \\ 3 & 6 & 8 \end{matrix}$$

Matrices are used extensively in many fields of computer science. For example, they are used to represent 2D and 3D transformations, the parameters of multivariate distributions, and datasets, though there are many more examples. Because matrices show up so often in advanced material, the ability to represent them and operate on them in programs is of great interest to developers. In fact, many hundreds of man-hours have already been spent writing efficient matrix libraries such as Eigen and GLM. With this project, you will hopefully gain an appreciation of the amount of work that goes into projects like these.

Generally, matrices can have any number of rows and any number of columns. Our matrices will be the 2D analog of the 1D **vector** ADT that was discussed in lecture. It will be a resizable collection of numbers of the same type that can efficiently be accessed by providing the coordinates `(r,c)` of a specific cell in the matrix. We will also provide several simple statistic operations (like **sum** and **product**) that tend to be used often in practice. The exact ADT interface will be described in Section 2.

The example matrix above stores integers, but that is not a general requirement. Our matrix class will work with any numeric type (e.g. **int**, **float**, **double**, etc.). We will accomplish this by using the **template** mechanism.

Representation

There are many possible ways to represent a matrix. For example, we could allocate a static 2D array of some maximum size (say **100 x 100**) and only make use of some of the cells. A **5 x 5** matrix, then, would take up a single **5 x 5** block in the top left corner of the 2D array. Resizing the matrix would be reasonably straightforward – we would simply have to extend the bounds of the block in the top left corner.

There are two significant limitations to this approach, however. 1) We waste a lot of space for small matrices. If our application required **3 x 3** or **4 x 4** matrices, for example, very little of the space that we allocated would actually be used to store important information. Most of it would go to waste. 2) There will always be some circumstances for which the static maximum size (**100 x 100** in our example) is *too small*. In Machine Learning, for example, it is not uncommon to work with matrices that have millions of rows and dozens of columns. With the static 2D array approach, we are not able to represent larger matrices at all.

We seem to have *conflicting* goals. We want to use as little memory as possible for small matrices, but we still want to have the ability to create large matrices when necessary. Thankfully, we *can* actually achieve both of those goals by using a **dynamically allocated array** instead of a statically allocated one. For each matrix, we only allocate exactly as much space as is really needed.

Dynamically allocating a 2D array in C++ can be done, but it is *very* messy. It requires dynamically allocating a 1D array of pointers to other 1D arrays, and then allocating a new 1D array for each element of the outer array. This approach also makes the program a bit slower (compared to the static 2D array approach), since we have to follow two different pointers to examine a single element of the matrix, and it is quite difficult to get the details right in practice.

A simpler alternative is to dynamically allocate a single **1D** array of size (**rows * cols**) instead. When we need to access a particular element, we use the `(r, c)` location in the 2D array to calculate the corresponding index in the 1D array. The figure below shows an example of how that process works. If we have the 2D matrix listed above, the first row in the 2D matrix uses the first 3 cells in the 1D array. The next row in the 2D array takes up cells 3-5 in the 1D array, and so on.

2D Array

	c = 0	c = 1	c = 2
r = 0	1	5	7
r = 1	2	4	0
r = 2	3	6	8

1D Array

0	1	2	3	4	5	6	7	8
1	5	7	2	4	0	3	6	8

The mapping from a 2D (r, c) pair to 1D index is straightforward:

$$Index(r, c) = r * cols + c$$

We could also do the reverse mapping, from a 1D index to a 2D `(r, c)` pair if we needed to. It requires using integer division and modulus instead of simple multiplication and addition, though. Thankfully, we won't need the reverse mapping for this application, but here are the formulas anyway:

$$r(Index) = \frac{Index}{cols}$$
$$c(Index) = Index \,\%\, cols$$

We will use the dynamically allocated 1D array approach in this assignment. You will allocate an array of size (**rows * cols**), and then use the first formula to look up the indices of cells using a given `(r, c)` pair whenever needed.

2. Design:

The design for this program will essentially be fixed. Your job will be to simply implement each of the methods that comprise the **Matrix** interface. Each method is explained in detail below:

1. **`Matrix();`**

   Default constructor – Should set `mRows` and `mCols` to 0 and `mData` to `NULL`. We have no data, so there is no use in dynamically allocating anything yet.

2. **`Matrix(const int rows, const int cols);`**

   Non-default constructor – Should save `rows` and `cols`. If both are non-zero, we need to allocate space for the matrix and initialize the cells to contain 0's. Otherwise, set `mData` to `NULL`.

3. **`Matrix(const Matrix<T>& orig);`**

   Copy constructor – Should copy `mRows` and `mCols` from the original. As with the non-default constructor, if both are non-zero, we need to allocate space for the matrix, but we will initialize the cells to contain the same values as the original instead of 0's. If either `rows` or `cols` is 0, set `mData` to `NULL`.

4. **`~Matrix();`**

   Destructor – Should deallocate any memory currently used by the matrix and set `mData` to NULL.

5. **`void print() const;`**

   Prints the matrix to the console. It would be a good idea to align elements within columns to make the matrix easy to read. You can use the `setw()` modifier in the `<iomanip>` library for that purpose.

6. **`void addRow();`**

   Adds a single row to the matrix. All cells in the new row should be set to 0, but the matrix should otherwise be unchanged.

7. **`void addCol();`**

   Adds a single column to the matrix. All cells in the new column should be set to 0, but the matrix should otherwise be unchanged.

8. **`void resize(const int rows, const int cols);`**

   Resizes the matrix to the given dimensions. If the matrix increases in size, all new cells should be set to 0, and all remaining cells should be unchanged. If the matrix decreases in size, the matrix should be truncated (as with the **Vector** ADT).

9. `T getCell(const int r, const int c) const;`

Returns the value at the given cell if `(r, c)` is within the bounds of the matrix. Otherwise, 0 should be returned.

10. `void setCell(const int r, const int c, const T& value);`

Place 'value' into the given cell `(r, c)` provided `(r, c)` is within the bounds of the matrix. Otherwise, ignore the set request.

11. `void fill(const T& value);`

Fill every cell in the matrix with the given value.

12. `T& operator()(const int r, const int c)`

Provided for you. Allows us to access individual cells using function notation. E.g.:

```
matrix(0, 0) = 5;
cout << matrix(2, 5) << endl;
```

13. `const T& operator()(const int r, const int c) const`

Provided for you. Allows us to read (but not write) individual cells using function notation. E.g.:

```
cout << matrix(2, 5) << endl;
```

14. `T sum(const int r1, const int c1, const int r2, const int c2) const;`

Returns the sum of all of the elements within the window [r1, c1] – [r2, c2]. All indices are inclusive.

15. `T product(const int r1, const int c1, const int r2, const int c2) const;`

Returns the product of all of the elements within the window [r1, c1] – [r2, c2]. All indices are inclusive.

16. `T max(const int r1, const int c1, const int r2, const int c2) const;`

Returns the largest of all of the elements within the window [r1, c1] – [r2, c2]. All indices are inclusive.

17. `double mean(const int r1, const int c1, const int r2, const int c2) const;`

Returns the average of all of the elements within the window [r1, c1] – [r2, c2]. All indices are inclusive.

18. `T sum() const;`

Returns the sum of all of the elements in the entire matrix.

19. `T product() const;`

Returns the product of all of the elements in the entire matrix.

20. `T max() const;`

Returns the largest of all of the elements in the entire matrix.

21. `double mean() const;`

Returns the average of all of the elements in the entire matrix.

22. `int getRows() const;`

Returns the number of rows in the matrix.

23. `int getCols() const;`

Returns the number of columns in the matrix.

3. Implementation:

There will be only a few implementation details that are required for this assignment:

1. All code relevant to the **Matrix** class **must** be contained within the `Matrix.h` header file. Your final program will consist of this header file and one `main.cpp` that will be provided for you. You are **NOT** to modify `main.cpp` or the class interface given in `Matrix.h`.
2. The implementations of each function in `Matrix.h` (except the two `operator()` functions that are provided) must be placed **outside** of the class definition. This will allow us to better maintain a separation between the *Matrix ADT interface* and its *implementation*.

As you write your code, you should compare your output to that provided in the file `expected_output.txt`. If yours varies from what is printed, there is a problem with your code that needs to be addressed. The test suite is written to only test 1 or 2 functions at a time, so if you encounter a discrepancy, it should not take long to figure out where something is going wrong.

NOTE: The test suite should give you a general idea if your code is working correctly, but no not trust it completely. It is entirely possible for your code to produce the same output, but still have a mistake somewhere.

Do everything you can to make your code “bullet-proof”. Try providing inputs that are egregiously wrong to see what your code does, for example. Stress tests like these are helpful for finding weaknesses and vulnerabilities. By eliminating them from the start, your code will be more robust and reliable.

4. Style

Make sure your code adheres to the guidelines provided in the Style Guide (available on Moodle). Your goal is to create code that is concise, descriptive, and easy for other humans to read. Avoid typos, spelling mistakes, or anything else that degrades the aesthetic of your code. Your final submission should be work that you are proud to call your own.

5. Testing:

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values, and save your testing output in text files for inclusion in your project report.

6. Documentation:

When you have completed your C++ program, write a short report using the project report template describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report to be submitted electronically.

7. Project Submission:

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic plagiarism analysis of all programs that are submitted.

When you have completed the tasks above go to Moodle to upload your documentation (a single **.pdf** file), and all C++ program files (**.h** and **.cpp**). Make sure your proof of testing is included in the documentation or is submitted as a separate file. Do NOT upload an executable version of your program.

The dates on your electronic submission will be used to verify that you met the due date above. Late projects will receive **NO** credit. You will receive partial credit for all programs that compile even if they do not meet all program requirements, so make sure to submit something before the due date, even if the project is incomplete.

8. Academic Honesty Statement:

Students are expected to submit their own work on all programming projects, unless group projects have been explicitly assigned. Students are NOT allowed to distribute code to each other, or copy code from another individual or website. Students ARE allowed to use any materials on the class website, or in the textbook, or ask the instructor for assistance.

This course will be using highly effective program comparison software to calculate the similarity of all programs to each other, and to homework assignments from previous semesters. Please do not be tempted to plagiarize from another student.

Violations of the policies above will be reported to the Provost's office and may result in a **ZERO** on the programming project, an **F** in the class, or suspension from the university, depending on the severity of the violation and any history of prior violations.