**CMPUT 275 – Tangible Computing II** **Winter 2021**

**Weekly Exercise #4**

---

**Matrix Class**

---

## Description

A *matrix* is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. When describing the size or dimension of a matrix, the number of rows comes before the number of columns. Below is an example of a 2 by 3 matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

We use the notation $A_{i,j}$ to denote the entry at the $i^{\text{th}}$ row and the $j^{\text{th}}$ column of matrix $A$. We use 0-indexing in this exercise, so the rows are numbered from 0 to $n-1$ and the columns from 0 to $m-1$ in an $n$ by $m$ matrix.

Your task is to implement a Matrix Class that works with the type `float`. The Matrix class must be declared in `matrix.h` and its implementation must be put in `matrix.cpp`. To internally represent the matrix, you should dynamically allocate a two dimensional array. You are not allowed to use STL containers for the internal representation of the matrix (e.g., std::vector).

In this exercise, **all operations (except the bracket operator and stream insertion/extraction) return a new instance of the Matrix class rather than modifying the existing object Also, to get full marks for style, you should use the const keyword when it is appropriate.** Your class should support the following operations:

- Matrix Addition
- Matrix Subtraction
- Matrix Multiplication
- Unary Negation
- Matrix Transposition Method
- Submatrix Method
- Bracket Operator
- ostream Insertion Operator (i.e., <<)
- istream Extraction Operator (i.e., >>)

Details about these operations can be found here.

All operations should be public methods (either as normal functions or as overloaded operators) of the Matrix class, except the insertion and extraction operator.

### Constructors

You must write two constructors for this class:

1. `Matrix(size_t num_rows, size_t num_columns, float init)`
2. `Matrix(size_t num_rows, size_t num_columns, float * arr_ptr)`

In both constructors, *num_rows* and *num_columns* specify the number of rows and columns of the matrix, respectively. The first constructor initializes all the elements to *init*, whereas the second constructor initialize the elements using the data stored in a one-dimensional array, which is pointed to by *arr_ptr*. This array will contain $num\_rows \times num\_columns$ elements.

### Matrix Addition, Subtraction, and Multiplication

Matrix addition, subtraction, and multiplication should be implemented by overloading the "+", "-", and "*" opeartor respectively.

```
Example: Matrix A = (B + C - D) * E;
```

### Unary Negation

The unary negation operator should be implemented for the Matrix class. When invoked, it should return a matrix where all the elements' signs are flipped.

```
Example: Matrix A = -B;
```

### Transposition

The transpose of an m by n matrix is the n by m matrix formed by turning rows into columns and vice versa. Matrix transpose is implemented as a public class method that takes no arguments. The function signature is: `Matrix transpose()`.

### Submatrix

The submatrix method returns a submatrix of the original matrix. It is a public method of the Mattrix class with the following signature:

```
Matrix submatrix(size_t row_start, size_t row_end,
                 size_t column_start, size_t column_end)
```

The returned submatrix should include rows from *row_start* to *row_end - 1* and columns from *column_start* to *column_end - 1*. You do not need to implement bounds checking for this method.

### Bracket Operator

The bracket operator should return a row of the matrix as a float pointer. Note that this is different from all the other methods that we've described so far, since it does not return a Matrix object. The argument *index* of the bracket operator specifies the row to return. This allows accessing and modifying of the matrix by chaining two bracket operators.

You do not need to implement bounds checking for this method; however, you are required to implement both the const version and the non-const version of this method. The following example demonstrates their usage:

```
Example:
Matrix A = Matrix(1, 1, 0.0);
const Matrix B = Matrix(1, 1, 1.0);
// non-const version called on A
// const version called on B
A[0][0] = B[0][0];
```

This sets the first element in the first row of A to 1.0.

### Insertion Operator

The insertion operator needs to be implemented for the Matrix class. When called, it should print elements row by row.

### Code Snippet #1

```
Matrix A = Matrix(2, 3, 1);
std::cout << A;
```

```
Output #1:
1 1 1
1 1 1
```

**Explanation**:
Elements within a row should be separated by spaces, and rows should be separated by new

lines. There should be no spaces at the end of each row, and there should be a new line after each row including the last row.

### Extraction Operator

The extraction operator reads $num\_rows \times num\_columns$ space separated elements from the input stream. Then, it populates elements of the matrix row by row.

### Code Snippet #2

```
Matrix A = Matrix(2, 3, 0.0);
std::cin >> A;
std::cout << A;
```

```
Input #2:
1 2 3 4 5 6
```

```
Output #2:
1 2 3
4 5 6
```

### Destructor and Copy Constructor

You are required to implement the destructor and the copy constructor for the Matrix class. The desturctor is responsible of making sure that all allocated memory is freed when a Matrix object goes out of scope. The copy constructor is responsible of making a deep copy of the Matrix class rather than a shallow copy. The following code snippet demonstrates a deep copy.

### Code Snippet #3

```
Matrix A = Matrix(1, 1, 0.0);
Matrix B = A;  // copy constructor is called here
B[0][0] = 100;
std::cout << A;
```

```
Output #3:
0
```

**Explanation**:
Because B is a deep copy of A, modifying B should not change A.

**Driver File**

A driver file (`main.cpp`) is provided to you where you can check if you implemented your methods with the correct function signatures. Do not include the driver file in your submission.

## Submission Guidelines:

Submit all of the following files as a `matrix.tar.gz` or `matrix.zip` file:

- `matrix.cpp`, the implementation of the matrix class
- `matrix.h`, the header file for the matrix class
- your `README`, following the Code Submission Guidelines
- a Makefile that has the following targets:
  - `matrix.o`: the first recipe which compiles the matrix class as an object file
  - `clean`: removes all executable and object files

  You may add additional targets if needed.

## Grading Comments:

- We will be testing your matrix class with a different driver. Make sure your function names and argument types are exactly what is required to make it work with the sample driver that we provided.
- To get full marks for style, you should use the const keyword and pass-by-reference for some operators when it is appropriate to do so.
- Style matters. Use appropriate comments, proper indentation, etc. Consult the style guide on eClass.
- We check your code for memory leaks. So it is important to free the memory when a Matrix object goes out of scope.
- Design choices are important when you implement the Matrix class. You might add private and public methods and attributes to this class only if they are necessary.