Breakout / Lab 04: **UPDATED**

Matrix Class & Operator Overloading

CSCI 1730 – Fall 2015

UPDATE: This breakout lab assignment has been updated! Please reread the document to identify the updated items. Here is a brief summary of the updates:

- Added an overloaded constructor that takes in an `std::initializer_list<std::initializer_list<double>>` or simply an `i_list` if you use the provided `typedef`. This change can be seen in the class prototype as well as the "Example Usage 1" section.
- Made some more member functions const-qualified where it made sense to do so. This change can be seen in the class prototype.
- Changed the matrix insertion operator overload to a matrix assignment operator overload. This change can be seen in the "Operator Overloading" section.

Problem / Exercise

Your goal for this breakout is to create an easy-to-use `Matrix` class in C++ that makes use of dynamic memory allocation and includes basic matrix operations both in the form of regular functions and via operator overloading. Here is the basic prototype for the `Matrix` class:

```
typedef unsigned int uint;
typedef initializer_list<initializer_list<double>> i_list;

class Matrix {

  public:

    Matrix(uint rows, uint cols);                  // constructor (all elements initialized to 0)
    Matrix(const i_list & list);                   // constructor (using initializer list)
    Matrix(const Matrix & m);                      // copy constructor
    ~Matrix();                                     // destructor

    Matrix add(double s) const;                    // add scalar to this matrix
    Matrix add(const Matrix & m) const;            // add this matrix and another matrix

    Matrix subtract(double s) const;               // subtract scalar from this matrix
    Matrix subtract(const Matrix & m) const;       // subtract another matrix from this matrix

    Matrix multiply(double s) const;               // multiply this matrix by a scaler
    Matrix multiply(const Matrix & m) const;       // multiply this matrix by another matrix

    Matrix divide(double s) const                  // divide this matrix by a scaler
    Matrix t() const;                              // transpose of this matrix

    const uint numRows() const;                    // returns the number of rows
    const uint numCols() const;                    // returns the number of cols

    double & at(uint row, uint col);               // get/set element at row,col
    const double & at (uint row, uint col) const;  // get element at row,col (when using a const object)

}; // Matrix
```

Important Class Details

Your `Matrix` implementation will contain elements of type `double`. In the prototype presented above, the term scaler refers to a regular number. For example, if you add a scaler to a matrix, then each element in the matrix gets that number added to it. Contrast this with the member functions that take a `Matrix` as their parameter. These functions represent regular matrix operations. For some of these operations (e.g., multiplication, transpose, etc.), you may need to consult some sort of reference in order to recall the exact procedure/meaning behind the operation.

NOTE: You **MAY** assume valid input for all operations.

NOTE: You **MAY NOT** use library classes such as `std::array`, `std::vector`, `std::list`, etc. for this breakout lab. You musr=t implement your `Matrix` class internally using a dynamically allocated array.

Example Usage 1

```
Matrix a(2, 2);
a.at(0, 0) = 1; // [ 1, 2 ]
a.at(0, 1) = 2; // [ 1, 3 ]
a.at(1, 0) = 1;
a.at(1, 1) = 3;

Matrix b(2, 1);
a.at(0, 0) = 3; // [ 3 ]
a.at(0, 1) = 2; // [ 2 ]

Matrix c = a.multiply(b);
cout << "[ " << c.at(0, 0) << " ]" << endl  // [ 7 ]
     << "[ " << c.at(1, 0) << " ]" << endl; // [ 9 ]

Matrix d = {{1, 2},  // this will implicitly call the overloaded constructor
            {3, 4}}; // that takes an initializer list
```

The usage of the member function `at(uint, uint)` is what facilities our ability to perform operations such as `a.at(0, 0) = 1`. If you implement this function carefully, then this behavior should work because the function returns a reference to an element. In order to support the constructor overload (i.e., matrix construction using an initializer list), you will need to use a standard template library (STL) class called `std::initializer_list`[1]. The type signature for the `<<` parameter representing the list should be `std::initializer_list<std::initializer_list<double>>` or simply `i_list` if you use the provided `typedef`

Operator Overloading

You will also need to overload operators in order to support the following functionality. It is up to you whether or not these should be member or non-member overloads.

```
// assume we have two matrices of appropriate size already set up
Matrix a;
Matrix b;

// after providing the overloads, you should be able to do any of the following operations
// using regular operators instead of the member functions
Matrix c0 = a + 5.2;
Matrix c1 = a + a;   // NOTE: these examples actually end up calling the copy constructor
Matrix c2 = a - 3.5;
Matrix c3 = b - b;
Matrix c4 = a * 2.1;
Matrix c5 = a * b;
Matrix c6 = a / 2.0;
```

[1] `std::initializer_list`: `http://en.cppreference.com/w/cpp/utility/initializer_list`

You should also support stream insertion (similar to overriding the `toString` method in Java) so that your matrices can easily be printed.

```
cout << a << endl; // example output: [ 1, 2 ]
                   //                 [ 1, 3 ]
```

You should also support matrix assignment using an initializer list (to easily overwrite existing elements) that looks like the following:

```
Matrix d(2, 2);
d = {{ 1, 2 },
     { 3, 4 }};
```

In order to support this last operator overload (i.e., matrix assignment using an initializer list), you will need to use a standard template library (STL) class called `std::initializer_list`. The type signature for the `<<` parameter representing the list should be `std::initializer_list<std::initializer_list<double>>` or simply `i_list` if you use the provided `typedef`. It is preferred that you specify the parameter as a `const i_list &` in order to avoid any unnecessary copying.

1 Group Brainstorm

You are NOT allowed to use the computers during this time.

Breakup into groups based on your seating and brainstorm about how to solve the problem or exercise. Make sure everyone understands the problem, and sketch out potential ways to move towards a solution. Perhaps something that was discussed during lecture might be useful?

2 Submit Individual Brainstorm

You may use a computer from this point forward.

Login to eLC and submit a version of your group's brainstorm, written in your own words. You may add additional information if you want. You need to write enough in order to convince the grader that you understand the problem or exercise and that you have a plan for moving forward towards a solution. Please include the last names of the other people in your group in your submission. The brainstorm submission should be available on eLC in your assignment dropbox. We prefer that you submit your individual brainstorms before the end of your breakout period, however, you generally have until 3PM on the day of your breakout (as indicated on eLC) to submit them.

NOTE: Submissions that do not include an individual brainstorm will not be graded.

3 C++ Code & Program

3.1 Setup

Make sure that all of your files are in a directory called `LastName-FirstName-lab04`, where `LastName` and `FirstName` are replaced with your actual last and first names, respectively.

3.2 Source Code Files

For this lab, you should organize your breakout lab into the following files:

- `Matrix.h`: This file should include the class prototype presented above as well as the prototypes for operator overloads that you implement. You **MAY NOT** modify the function prototypes that are included in the `Matrix` class prototype. However, you may add additional function prototypes and variables to the class prototype as needed. Make sure that this header file also includes a header guard (i.e., the `#ifndef` macro, etc.).
- `Matrix.cpp`: This file should contain the implementation of your class's functions as well as the implementation of any operator overloads that you implement.
- `lab04.cpp`: This file should contain a small/moderately sized driver that demonstrates the full range of functionality of your `Matrix` class.

Additionally, make sure that you adhere to the following:

- All functions must be documented using Javadoc-style comments. Use inline documentation, as needed, to explain ambiguous or tricky parts of your code.

3.3 Makefile File

You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. Make sure that your `Matrix.cpp` file compiles to `Matrix.o`. This is very important because we will be testing your submission by linking against your `Matrix.o` file. The resulting executable should be called `lab04`.

Make sure that when you compile, you pass the following options to `g++` in addition to the `-c` option:

```
-Wall -std=c++11 -g -O0 -pedantic-errors
```

Here is a link to the `Makefile` for the "gradebook" example we coded up in class that you might find useful as a reference point: `https://gist.github.com/mepcotterell/45a10fa72b208a76d968`.

3.4 README File

Make sure to include a `README` file that includes the following information presented in a reasonably formatted way:

- Your Name and 810/811#
- Instructions on how to compile and run your program.
- A Reflection Section. In a paragraph or two, compare and contrast what you actually did to complete the problem or exercise versus what you wrote in your initial brainstorm. How will this experience impact future planning/brainstorms?

Here is a partially filled out, example `README` file: `https://gist.github.com/mepcotterell/3ce865e3a151a3b49ec3`.

NOTE: Try to make sure that each line in your `README` file does not exceed 80 characters. Do not assume line-wrapping. Please manually insert a line break if a line exceeds 80 characters.

3.5 Compiler Warnings

Since you should be compiling with both the `-Wall` and `pedantic-error` options, your code is expected to compile without `g++` issuing any warnings. For this lab, compiling without warnings will be one or more of the test cases.

3.6 Memory Leaks

Since this breakout lab makes use of dynamic memory allocation, you are expected to ensure that your `Matrix` implementation doesn't result in any memory leaks. We will test for memory leaks using the `valgrind` utility. For this lab, having no memory leaks will be one or more of the test cases.

4 Submission

Before the day of the next breakout session, you need to submit your code. You will still be submitting your project via nike. Make sure your work is on `nike.cs.uga.edu` in a directory called `LastName-FirstName-lab04`. From within the parent directory, execute the following command:

```
$ submit LastName-FirstName-lab04 cs1730a
```

It is also a good idea to email a copy to yourself. To do this, simply execute the following command, replacing the email address with your email address:

```
$ tar zcvf LastName-FirstName-lab04.tar.gz LastName-FirstName-lab04
$ mutt -s "lab01" -a LastName-FirstName-lab04.tar.gz -- your@email.com < /dev/null
```