EECS 268: Spring 2009
Laboratory 2: Systems of Linear Equations

Due: 11:59:59 pm the day before your lab meets the week of February 16

Lecture Topics: Classes, Exceptions
Lab Topics: Constructors/Destructors, Dynamic Memory Management

1. Introduction

You have become a victim of your own success! Your Matrix class has proved to be so popular with your clients that they have asked you to expand upon the program.

(This lab builds upon and extends lab 1. You *must* turn in a working lab 1 for grading before starting on this lab. Completing lab 2 cannot be used in lieu of completing and submitting lab 1!)

Your clients have requested three changes to your program. First, they want your program to work for matrices of *any* size, with the size of the *n x n* matrix being specified by the user at runtime. Second, they want your program to provide helpful warning messages if the user attempts to perform any invalid operation, such as attempting to multiply matrices with incompatible sizes. Third, they want your program to use the existing functionality of your Matrix class (as well as the new functionality of a Vector class) to automatically solve systems of linear equations.

Through this lab you will demonstrate the ability to identify, formulate, and implement software solutions for various Matrix computational problems.

2. Getting Started

2.1. Dynamic Memory Management

Since the size of your matrix arrays can no longer be limited to a predefined constant (such as `MaxSize` in Lab 1), you will be required to allocate and deallocate memory for your arrays at runtime. To do this, we will use the constructors and destructors in the Matrix class to keep track of the memory as it is allocated and deallocated.

Additionally, because C++ does not automatically copy any object's data that was allocated at runtime, we will be required to define our own copy constructors and assignment operators to prevent memory leaks and corrupted memory.

In the Matrix class, we will use the following function prototypes for the constructors, the destructor, and the overloaded assignment operator:

```
Matrix();                               // Default constructor
Matrix(int n);                          // Custom constructor
Matrix(const Matrix& m);                // Copy constructor
Matrix& operator=(const Matrix& m);     // Overloaded assignment operator
~Matrix();                              // Destructor
```

The constructors will be responsible for allocating memory for a two dimensional array of doubles, based either on the integer *n* or the size of the Matrix *m*. (In the case of the default constructor, the size of the matrix should be initialized to 0 and no memory should be allocated.) The destructor will be responsible for deallocating memory for the two dimensional array of doubles. The assignment operator will be responsible for first deallocating any previously allocated memory and then allocating new memory based on the size of the Matrix *m*. (For example, the assignment in the following code has three effects:

```
Matrix m(3), n(10);
m = n;
```

First, the memory for the original 3*x*3 array is deallocated. Second, the memory for a new 10*x*10 array is allocated. Finally, the data is copied from *n*'s 10*x*10 array into *m*'s 10*x*10 array.)

Similarly, you will define a new Vector class with same dynamic memory requirements for a one dimensional array of doubles. In the Vector class, we will use the following function prototypes for the constructors, the destructor, and the overloaded assignment operator:

- `Vector();` `// Default constructor`
- `Vector(int n);` `// Custom constructor`
- `Vector(const Vector& v);` `// Copy constructor`
- `Vector& operator=(const Vector& v);` `// Overloaded assignment operator`
- `~Vector();` `// Destructor`

2.2. Exception Handling

Your program will be expected to gracefully recover (i.e., display a helpful warning message and continue processing) from the following runtime errors:

- An attempt to invert a matrix whose determinant is zero.
- An attempt to multiply two matrices with incompatible sizes.
- An attempt to multiply a matrix and a vector with incompatible sizes.
- An attempt to compute an invalid sub matrix.

To do this, you will define a new exception class, MatrixException. Your program will be expected to throw this exception from within the Matrix class whenever you detect one of the above situations. Be sure to modify the Matrix function prototypes as follows:

- `double determinant() const;`
- `Matrix inverse() const throw (MatrixException);`
- `Matrix subMatrix(int i, int j) const throw (MatrixException);`
- `Matrix operator*(const Matrix &rhs) const throw (MatrixException);`
- `Vector operator*(const Vector &rhs) const throw (MatrixException);`

All exceptions must be caught and recovered from in your main function. Your main function will be expected to print out a short warning message describing the error and to continue processing the input file.

2.3. Solving Linear Systems of Equations

We will test the functionality of the Matrix and Vector classes by solving systems of *n* linear equations with *n* unknowns. If **M** is an *n x n* matrix and **v** is an vector of length *n*, it can represent the linear system:

$$\mathbf{Mx} = \mathbf{v}$$

where **x** is a vector of *n* unknowns. The equation is solved by inverting **M** and multiplying it on the left of **v**:

$$\mathbf{x} = \mathbf{M}^{-1}\mathbf{v}$$

We will implement this functionality by defining a new Vector class. In addition to the function prototypes defined above, the Vector class will be required to include the overloaded input and output operators. Because all data in the Vector and Matrix classes is expected to be private, the Vector class will also require getter and

setter functions for the size of the Vector and for the elements in the vector. For example, the vector class may define two functions, *int getElement(int i)* and *void setElement(int i, double value)*, that get and set the *i*-th element of the vector. Note: Declare these getter and setter functions using `const` wherever possible (except on primitive data types).

In addition to defining the Vector class, you will also need to define a new overloaded multiplication operator for the Matrix class that will allow for the multiplication of a Matrix and a Vector:

- `Vector operator*(const Vector &rhs) const throw (MatrixException);`

3. Development and Testing

Your program is to read an input file consisting of the following three commands: matrix, vector, and solve.

matrix *n m00 ... m0n ... mn0 ... mnn*

- The matrix command will always be immediately followed by a matrix definition. You will need to read in this matrix, but no further processing or output will be required for this command.

vector *n v0 ... vn*

- The vector command will always be immediately followed by a vector definition. You will need to read in this vector, but no further processing or output will be required for this command.

solve

- The solve command will require your program to attempt to solve a system of linear equations using the last matrix and vector that were read. (Do not try to detect invalid requests in your main program. All problems are to be detected in your `Matrix` methods which will throw exceptions as needed. Your client code must properly catch, report, and recover from all exceptions.) If an equation is successfully solved, your program must print (i) the solution vector **x**, (ii) the product **M*x**, and (iii) the vector **v**. (Of course, the output generated by (ii) and (iii) should be the same!)

Note: There may be additional whitespace (newlines, etc.) in the input. However, the order of the matrix and vector specifications will not change.

4. Input & Output

Your program will be expected to read input from the first file specified on the command line and to write output to the second file specified on the command line. Your program should display an error message and exit immediately if the expected command-line parameters are not provided. Following is an example of how your program will be run from the command line:

```
$ ./main input.txt output.txt
```

Your program will continue reading and processing the commands until you reach the end of the input file. After processing each command, your program should write the result to the output file. All input and output should use C++ streams (>> and <<).

To make sure the format of this file is clear, you can reference the following sample input. Note, however, that your program will be tested with other files.

4.1. Input File

```
vector 3
     1
```

```
     2
     3

matrix 3
     1     2     3
     3     2     1
     1     3     2

solve

vector 4
     1
     2
     3
     4

solve
```

4.2. Output File

```
x =
-0.0833333
1.41667
-0.583333

M * x =
1
2
3

v =
1
2
3

Error: Invalid sizes for matrix/vector multiplication.
```

5. Grading Criteria

Grades will be assigned according to the following criteria:

20	**Solving Linear Systems of Equations** - Implemented all necessary functions in Matrix and Vector class. - Correctly use these classes in the main function to solve linear systems of equations.
25	**Dynamic Memory Management** - Constructors, destructor, and assignment operator implemented correctly in Matrix and Vector.
15	**Exception Handling** - Correctly defined a new MatrixException class. - Exceptions are thrown for each case described above. - All exceptions are caught and recovered from in the main function.
30	**Documentation** - Javadoc comments for all files, classes, and functions. - Internal documentation (especially recursive functions).
10	**Programming Style & Output Formatting** - Use standard paragraphing conventions and object orientated design practices.
100	**Total**