ECE2036 **Fall Semester, 2012**

Project 3 – Matrix Calculator

Assigned: September 17, 2012 *Due: September 28, 2012, 11:59pm*

Introduction. We will extend our previously developed *SimpleCalculator* to provide for the simple matrix operations add, subtract and multiply (we won't do divide). To implement this, we will create a class called `Matrix` that implements a two–dimensional matrix. We will implement several constructors, including one that populates the matrix with data from a character string. To describe a matrix with a string, we use parenthesis to delineate the rows of the matrix. For example:
`(1,2,3),(4,5,6),(7,8,9)`
would represent the matrix:

```
1 2 3
4 5 6
7 8 9
```

All the rules from previous calculator program apply, such as only a single operator per input line, lines starting with an operator use the prior result at the first operand, etc. We will also use a *Not A Matrix* flag in our *matrix* class to indicate that the matrix is invalid. This would be set when the size of the matrices being added or multiplied are not compatible or when the matrix is unintialized.

Since we don't know at compile time how many rows or columns a matrix object will have, we must use dynamic memory management to allocate and return memory for the matrix elements. For simplicity in this assignment, we will simply allocate a one dimensional array of elements of size "Number Rows * Number Columns". The memory should be allocated in the matrix constructor if the size is known. The memory should also be freed in the matrix destructor.

Specific Program Requirements.

1. You must define and implement a `Matrix` class, with a constructor with a `string` argument, to construct a matrix with initial contents. In this case the size of the matrix is apparent from the input string. Additionally, you will need a second constructor with two unsigned int's specifying an m x n matrix initialized with all zeros (to be used for results in arithmetic expressions).
2. Since your `Matrix` class allocates memory in the constructor (for the matrix elements), you *MUST* implement a destructor that frees the memory, plus a copy constructor and an assignment operator.
3. You must implement a `IsNaM` function that returns a boolean true/false indicating whether the matrix is *Not a Matrix*. As mentioned above, a matrix is *Not a Matrix* if it is the result of arithmetic operators of incorrect size.
4. The `Matrix` class must implement *indexing operator* (`operator[]`) to access individual elements in the matrix. Think carefully about how to do this.
5. All operator overloads must be implemented as **member functions**, and all operator overloads must have `const` for both left-hand-side and right-hand-side.
6. Values within the matrix should be stored as type `Element_t`. This is defined to be an `int` in `matrix.h`, but we could change to a `double` later and not require any code changes.
7. Matrix arithmetic should be performed using operator overloading.
8. You must implement a `Print()` member function. The printed matrix must have the column values aligned right–justified, such as:

```
25  123   5   0   0
 0    2   3   0   0
 0  999  10   0   0
 0    0   0   1   1
 3    0   0   0   3
```

To make the assignment a bit easier, you can assume that no individual element value has more than 4 digits and all numbers are positive.

9. You will likely find that your main loop in `matrix-calc.cc` is nearly identical to the main loop in the original `complex-calc.cc` from the earlier program.

Design Philosophy. Your program design should be as simple as possible, but no simpler. [1]

Resources. There are a number of files that are given to you. These are all on the `jinx-login` system, and you will copy these to your own directory (instructions as to how to do this are below).

1. A skeleton `matrix-calc-skeleton.cc` program that is a starting point. It contains the subroutine to read a line from standard in and call the string parser. Your main loop should exit when an empty line is encountered (when the string parser returns a count of zero).
2. Skeleton `Matrix.h` describes the requirements for the `Matrix` class. You should implement the required functions in `Matrix.cc`.
3. `string-parse.h` and `string-parse.cc` are provided for string parsing. Note the *ToElement* conversion function that converts a string to an integer `Element_t`. Also note that **anything inside a pair of parens is not a separator**. Thus, parsing "`(1,2,3),(4,5,6)`" with a separator of comma gives two substrings, "`(1,2,3)`" and `(4,5,6)`". To reduce those substrings to three individual integers you of course use `RemoveParens` and then `SringParse` again.
4. A `Makefile` for building the executable binary.
5. A second executable called `test-matrix` runs several canned test cases.
6. `test-matrix-out.txt` is the expected results from running `test-matrix`.

Your program can be compiled and tested on any available computing platform that has a C++ compiler. The instructor and TA will compile and test your program on the jinx linux systems. Be sure to put your name on the source code in the comments section.

Copying the Project Skeletons

1. Log into `jinx-login.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE2036 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE2036/MatrixCalculator .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `MatrixCalculator`

   ```
   cd MatrixCalculator
   ```

4. Copy the provided `matrix-calc-skeleton.cc` to `matrix-calc.cc` as follows:

[1]Paraphrased from quote by Albert Einstein

```
cp matrix-calc-skeleton.cc matrix-calc.cc
```

5. At this point, you can run the `make` program to compile the skeletons. Of course it won't do anything useful since you have not implemented your `Matrix` class.
6. Use `emacs` or `vi` to edit the files as needed to implement your program.
7. You can test your program against a fixed set of inputs in file `input.txt` as follows:

   ```
   ./matrix-calc < input.txt
   ```

8. The expected outputs from this are found in file `output.txt`

Checking for Memory Leaks and Other Memory Problems This type of program with dynamic memory management is particularly difficult to get implemented correctly. Often programs that are *buggy* and mis-manage memory will in fact work properly and produce right answers in some cases. However, we still need to correctly manage all dynamically allocated memory. On the linux systems, there is a nice tool called `valgrind` that helps us identify and fix problems with memory management.

To use valgrind, you simply run the valgrind program passing as arguments the name of the program to be tested and all paramenters to your program. For the matrix calculator, you would likely say:

```
/opt/valgrind-3.7.0/bin/valgrind --tool=memcheck ./matrix-calc < input.txt
```

The output of valgrind is sometimes hard to read; so we will go over this in class. We *will* run valgrind on your program when grading, and to get a 100% you must get a "Clean Valgrind" (with no errors reported).

Turning in your Project. The system administrator for the jinx cluster has created a script that you are to use to turn in your project. The script is called `riley-turnin` and is found in `/usr/local/bin`, which should be in the search path for everyone. From your **home directory** (not the MatrixCalculator subdirectory), enter:

```
riley-turnin MatrixCalculator.
```

This automatically copies everything in your `MatrixCalculator` directory to a place that I can access (and grade) it.