1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework04, put each programming exercise into its own subdirectory of this directory, zip the entire Homework04 directory up into the file Homework04.zip, and then submit this zip file to Homework #4.

Make sure that you:

- Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
- Check the contents of the zip file before uploading it. Make sure all the files are included.
- Make sure that the file was submitted correctly to MyCLasses.

All class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. No inline coding in the .h files. In addition you must create a make file that compiles and links the project on a Linux computer with a Debian or Debian branch flavor.

2 Programming Exercises

1. This exercise is on the creation of a Point class, utilizing operator overloading. I am going to leave the way you store the data up to you. As we discussed in class, the concept of data hiding is that someone who uses your class structure need not know (and should not need to deal with) how you are storing the data. The interface functions should take care of all of this. We will discuss the mathematics involved (not very much here). The test program below will show you what needs to be implemented and what operators need to be overloaded. It will also show you the structure of the returned vales as well as the formatting.

   The class structure is to be named `Point`. It represents a point in three dimensions, that is (x, y, z). The values of x, y, and z are all decimal values, hence a double data type for these three entries will do here.

   - The addition of two points is another point and is done by adding the respective x, y, and z values. So in mathematical notation,

     $$(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

   - The subtraction of two points is another point and is done by subtracting the respective x, y, and z values. So in mathematical notation,

     $$(x_1, y_1, z_1) - (x_2, y_2, z_2) = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$$

- Multiplying a number and a point (called scalar multiplication) is just multiplying each entry by that number. So in mathematical notation,

$$a \cdot (x_1, y_1, z_1) = (a \cdot x_1, a \cdot y_1, a \cdot z_1)$$

  This can also be done on either side of the point, that is,

$$(x_1, y_1, z_1) \cdot a = (a \cdot x_1, a \cdot y_1, a \cdot z_1)$$

- We can also divide a point by a number by dividing each entry by the number. So in mathematical notation,

$$(x_1, y_1, z_1)/a = (x_1/a, y_1/a, z_1/a)$$

- The length of a point is just the distance from the point to the origin, specifically, $\sqrt{x^2 + y^2 + z^2}$.
- The multiplication of two points will be interpreted as what is called the dot product (or scalar product). Specifically, if the two points are $p = (x_1, y_1, z_1)$ and $q = (x_2, y_2, z_2)$, then $p * q = x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2$. Note that the product of two points is not a point but a decimal number.
- Two points are equal if their x, y, and z coordinates are all equal to each other. Specifically, $(x_1, y_1, z_1) = (x_2, y_2, z_2)$ means that $x_1 = x_2$ and $y_1 = y_2$ and $z_1 = z_2$.

Given the following testing program,

```cpp
#include <iostream>
#include "Point.h"

using namespace std;

int main() {
    Point p1;
    Point p2(3, 5, 1);
    Point p3(-2, 4, 7);

    cout << p1 << "  " << p2 << "  " << p3 << endl;

    Point p4 = p2;
    cout << p4 << endl;

    p4 = p3;
    cout << p4 << endl;

    p4 = p2 + p3;
    cout << p4 << endl;

    p4 = p2 - p3;
    cout << p4 << endl;

    double d = p2 * p3;
    cout << d << endl;

    p4 = p2;

    if (p2 == p4)
```

```
        cout << "Points are equal." << endl;
    else
        cout << "Points are different." << endl;

    cout << p4.length() << endl;

    cout << p4 << endl;
    cout << p4.getX() << endl;
    cout << p4.getY() << endl;
    cout << p4.getZ() << endl;

    p4.setX(-5.2);
    p4.setY(7.1);
    p4.setZ(3.5);

    cout << p4 << endl;
    cout << p4.getX() << endl;
    cout << p4.getY() << endl;
    cout << p4.getZ() << endl;

    p4.setXYZ(3.14159, 2.718281828, 1.618033988);
    cout << p4 << endl;

    cout << p3 << endl;
    cout << 3 * p3 << endl;
    cout << p3 * 2 << endl;
    cout << p3 / 4 << endl;

    return 0;
}
```

The output would be

```
(0, 0, 0)  (3, 5, 1)  (-2, 4, 7)
(3, 5, 1)
(-2, 4, 7)
(1, 9, 8)
(5, 1, -6)
21
Points are equal.
5.91608
(3, 5, 1)
3
5
1
(-5.2, 7.1, 3.5)
-5.2
7.1
3.5
(3.14159, 2.71828, 1.61803)
(-2, 4, 7)
(-6, 12, 21)
(-4, 8, 14)
(-0.5, 1, 1.75)
```

2. This exercise is a continuation of Lab #4 and the last homework assignment on dynamically allocated two-dimensional arrays. You may want to consult Lab #4 and the last homework to refresh your memory on the setup of the Array2D class structure as well as its use of dynamic memory.

(a) Take the Array2D class from Lab #4 and change the name of the class to Matrix. The array (double pointer) should be changed from integers to an array of doubles, the rows and cols should remain as is. If you had trouble with any part of Lab #4 you can use the published solutions for the lab as a template to get started.

(b) Add the following to the Matrix class.

- A Copy Constructor for the class.
- Overloaded assignment statement for the class.
- A display function that takes in a single integer parameter that specifies the width of the column. The display of the matrix will be right justified in the column.
- Overloaded + operator. Addition of two matrices can only be done if the number of rows and the number of columns between the two are equal. So you can add two 3×5 matrices or two 7×4 matrices but you cannot add a 3×5 matrix to a 7×4 matrix. If the sizes are equal then the sum is another matrix of the same size and each entry is the sum of the corresponding entries. For example,

$$\begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 3 & 6 & 7 \\ 0 & 5 & 0 & 9 & 1 \\ 1 & 7 & 0 & 6 & 2 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 10 & 8 & 11 \\ 2 & 9 & 0 & 15 & 9 \\ 8 & 9 & 9 & 14 & 10 \end{bmatrix}$$

  If the sizes are not compatible have the function return a 1×1 matrix with its only entry being 0.
- Overloaded $-$ operator. Subtraction of two matrices can only be done if the number of rows and the number of columns between the two are equal. So you can subtract two 3×5 matrices or two 7×4 matrices but you cannot subtract a 3×5 matrix from a 7×4 matrix. If the sizes are equal then the difference is another matrix of the same size and each entry is the difference of the corresponding entries. For example,

$$\begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix} - \begin{bmatrix} 7 & 8 & 3 & 6 & 7 \\ 0 & 5 & 0 & 9 & 1 \\ 1 & 7 & 0 & 6 & 2 \end{bmatrix} = \begin{bmatrix} -3 & -4 & 4 & -4 & -3 \\ 2 & -1 & 0 & -3 & 7 \\ 6 & -5 & 9 & 2 & 6 \end{bmatrix}$$

  If the sizes are not compatible have the function return a 1×1 matrix with its only entry being 0.
- Overload the $*$ operator so that you can take a double times a Matrix. For example if `M` is a matrix then the syntax `2.4 * M` would call this function. Note that this will take a friend function in a similar manner to how the RationalNumber class did the same thing. When you take a number times a matrix you multiply all of the entries in the matrix by the number. For example,

$$3 \cdot \begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 12 & 12 & 21 & 6 & 8 \\ 6 & 12 & 0 & 18 & 24 \\ 21 & 6 & 27 & 24 & 24 \end{bmatrix}$$

- Overload the $*$ operator so that you can take a Matrix times a double. For example if `M` is a matrix then the syntax `M * 2.4` would call this function. Note that this does not take a friend function. Again use the RationalNumber class code as a template. When you take a matrix times a number you multiply all of the entries in the matrix by the number. For example,

$$\begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix} \cdot 3 = \begin{bmatrix} 12 & 12 & 21 & 6 & 8 \\ 6 & 12 & 0 & 18 & 24 \\ 21 & 6 & 27 & 24 & 24 \end{bmatrix}$$

- Overload the stream out operator << to print the matrix (array) on one line. The rows are to have brackets around them and the entire matrix is to have brackets around it. The entries are to be separated by a single space. For example, if the matrix `M` is

$$\begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix}$$

  then `cout << M;` would print `[[4 4 7 2 4][2 4 0 6 8][7 2 9 8 8]]`.

- Overload the array access operator `[]`. This should return a pointer to the indexed row. So the syntax `M[2]` will return a pointer to the row (array) at index 2. So in our last example, `M[2]` will be a pointer to the row `[7 2 9 8 8]`. Do range checking on this so that if the row index is out of range the function will return `nullptr` instead of a memory address. A note on this operator overload. Since the return type is a pointer to the row array, the syntax `M[2][3]` will automatically access (and update if on the left of an assignment statement) the $(2, 3)$ position in the array. The only downside is that with this syntax (unlike the set command) will not check the column index for validity.

- Create a transpose function that will return the transpose of the current matrix. The transpose is where the rows and columns are switched. So the first row of a matrix is the first column of the transpose, the second row of a matrix is the second column of the transpose, and so on. For example, the transpose of

$$\begin{bmatrix} 4 & 4 & 7 & 2 & 4 \\ 2 & 4 & 0 & 6 & 8 \\ 7 & 2 & 9 & 8 & 8 \end{bmatrix}$$

  is

$$\begin{bmatrix} 4 & 2 & 7 \\ 4 & 4 & 2 \\ 7 & 0 & 9 \\ 2 & 6 & 8 \\ 4 & 8 & 8 \end{bmatrix}$$

- **Optional Extra Credit:** Overload the $*$ yet again but this time for multiplying two matrices together. So if `A` and `B` are two matrices the syntax `A * B` will call this function. To multiply two matrices the first thing is that the number of columns of the first must be the same as the number of rows of the second. So if `A` is $n \times m$ and `B` is $m \times k$ you can multiply the two matrices together. If the number of columns of the first and the number of rows of the second do not match then the multiplication is not defined. If it is defined then the result is a matrix `C` that has size $n \times k$. The entries of the product matrix are defined as follows. Denote A and B as follows,

  $$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2m} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nm} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1k} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2k} \\ b_{31} & b_{32} & b_{33} & \cdots & b_{3k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \cdots & b_{mk} \end{bmatrix}$$

  then

  $$AB = C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1k} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2k} \\ c_{31} & c_{32} & c_{33} & \cdots & c_{3k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \cdots & c_{nk} \end{bmatrix}$$

  with

  $$c_{ij} = \sum_{t=1}^{m} a_{it}b_{tj} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{im}b_{mj}$$

  In words, take the i^{th} row of A, and the j^{th} column of B, they are the same length. Multiply the respective entries, then add all those products up. As for the computer function, if the sizes are not compatible have the function return a 1×1 matrix with its only entry being 0.

Given the following testing program, the output is below. Note that if you do not do the extra credit you should comment out the matrix multiplication on line 121.

```
1 #include <cstdlib>
2 #include <ctime>
```

```cpp
#include <iostream>

#include "Matrix.h"

using namespace std;

int main() {
    srand(time(0));

    Matrix A(3, 5, 3.14);
    Matrix B = A;
    Matrix C;

    A.display(7);
    cout << endl;

    for (int i = 0; i < A.getRows(); i++)
        for (int j = 0; j < A.getCols(); j++)
            A.set(i, j, (rand() % 1000) / 10.0);

    A.display(7);
    cout << endl;

    A.set(10, 3, -15);
    A.display(7);
    cout << endl;

    A.set(1, 3, -15);
    A.display(7);
    cout << endl;

    cout << A.get(2, 2) << endl;
    cout << A.get(2, 20) << endl;
    cout << endl;
    B.display();
    cout << endl;

    C = A = B;

    A.display(7);
    cout << endl;
    C.display(6);
    cout << endl;

    for (int i = 0; i < B.getRows(); i++)
        for (int j = 0; j < B.getCols(); j++)
            B.set(i, j, rand() % 10);

    for (int i = 0; i < C.getRows(); i++)
        for (int j = 0; j < C.getCols(); j++)
            C.set(i, j, rand() % 10);

    B.display(7);
    cout << endl;
    C.display(7);
    cout << endl;

    A = B + C;

    A.display(7);
    cout << endl;

    A = B - C;

    A.display(7);
    cout << endl;

    for (int i = 0; i < A.getRows(); i++)
        for (int j = 0; j < A.getCols(); j++)
            A.set(i, j, (rand() % 1000) / 10.0);

    A.display(7);
    cout << endl;

    B = 3 * A;
    B.display(7);
    cout << endl;

    B = A * 7;
    B.display(7);
    cout << endl;

    B = 2 * A - 7 * C;
    B.display(7);
    cout << endl;

    A.display(7);
    cout << endl;

    A[2][2] = 123;
    A[1][3] = -15;
    A[0][4] = -25;

    A.display(7);
    cout << endl;
    cout << A[0][1] << " " << A[1][4] << " " << A[2][3] << " " << endl;

    // cout << A[10][11] << endl;  // Error

    cout << A << endl << endl;

    Matrix D(3, 4);
    Matrix E(4, 2);

    for (int i = 0; i < D.getRows(); i++)
        for (int j = 0; j < D.getCols(); j++)
            D[i][j] = rand() % 11 - 5;

    for (int i = 0; i < E.getRows(); i++)
        for (int j = 0; j < E.getCols(); j++)
            E[i][j] = rand() % 11 - 5;

    D.display(7);
    cout << endl;

    E.display(7);
    cout << endl;

    B = D * E;
    B.display(7);
    cout << endl;

    cout << B << endl << endl;

    B = B.transpose();
    B.display(7);
    cout << endl;

    cout << B << endl << endl;

    return 0;
}
```

The output would be

```
   3.14    3.14    3.14    3.14    3.14
   3.14    3.14    3.14    3.14    3.14
   3.14    3.14    3.14    3.14    3.14

     66    16.1    71.2    79.1    63.7
   65.9     1.2    91.3    67.1    98.2
   24.2      35    72.8    54.9      85

Index out of bounds.
     66    16.1    71.2    79.1    63.7
   65.9     1.2    91.3    67.1    98.2
   24.2      35    72.8    54.9      85

     66    16.1    71.2    79.1    63.7
   65.9     1.2    91.3     -15    98.2
   24.2      35    72.8    54.9      85

72.8
Index out of bounds.
0

3.14 3.14 3.14 3.14 3.14
3.14 3.14 3.14 3.14 3.14
3.14 3.14 3.14 3.14 3.14

   3.14    3.14    3.14    3.14    3.14
   3.14    3.14    3.14    3.14    3.14
   3.14    3.14    3.14    3.14    3.14

  3.14   3.14   3.14   3.14   3.14
  3.14   3.14   3.14   3.14   3.14
  3.14   3.14   3.14   3.14   3.14

      0       1       8       2       6
      9       4       2       6       8
      5       0       9       7       4

      9       7       7       3       0
      7       3       4       0       6
      8       2       8       8       4

      9       8      15       5       6
     16       7       6       6      14
     13       2      17      15       8

     -9      -6       1      -1       6
      2       1      -2       6       2
     -3      -2       1      -1       0

   28.8    63.9    26.5    50.9    20.3
   71.1    64.8    13.7    17.6    32.6
   96.7     4.1    98.9     3.7      56

   86.4   191.7    79.5   152.7    60.9
  213.3   194.4    41.1    52.8    97.8
  290.1    12.3   296.7    11.1     168
  201.6   447.3   185.5   356.3   142.1
  497.7   453.6    95.9   123.2   228.2
  676.9    28.7   692.3    25.9     392

   -5.4    78.8       4    80.8    40.6
   93.2   108.6    -0.6    35.2    23.2
  137.4    -5.8   141.8   -48.6      84

   28.8    63.9    26.5    50.9    20.3
   71.1    64.8    13.7    17.6    32.6
   96.7     4.1    98.9     3.7      56

   28.8    63.9    26.5    50.9     -25
   71.1    64.8    13.7     -15    32.6
   96.7     4.1     123     3.7      56

63.9 32.6 3.7
[[28.8 63.9 26.5 50.9 -25][71.1 64.8 13.7 -15 32.6][96.7 4.1 123 3.7 56]]

      4      -1       3       4
     -3      -5      -1       2
      3      -5       3       2

     -2      -5
      5      -4
      1      -5
      3       0

      2     -31
    -14      40
    -22     -10

[[2 -31][-14 40][-22 -10]]

      2     -14     -22
    -31      40     -10

[[2 -14 -22][-31 40 -10]]
```