```cpp
#include <iostream>
#include "matrix.h"
using namespace std;

int main()
{
    Matrix matrix1(3, 3);
    Matrix matrix2(3, 3);
    Matrix matrix3(2, 3);
    Matrix matrix4(3, 2);

    cout << "Matrix 1" << endl;
    double num = 0;
    for (int i = 0; i < 3; i++) // Matrix1 Population
    {
        for (int j = 0; j < 3; j++)
        {
            matrix1[i][j] = num;
            num += 1;
        }
    }
    cout << matrix1 << endl;

    cout << "Matrix 2" << endl;
    num = 8;
    for (int i = 0; i < 3; i++) // Matrix2 Population
    {
        for (int j = 0; j < 3; j++)
        {
            matrix2[i][j] = num;
            num -= 1;
        }
    }
    cout << matrix2 << endl;

    cout << "Matrix 3" << endl;
    num = 0;
    for (int i = 0; i < 2; i++) // Matrix3 Population
    {
        for (int j = 0; j < 3; j++)
        {
            matrix3[i][j] = num;
            num += 1;
        }
    }
    cout << matrix3 << endl;

    cout << "Matrix 4" << endl;
    num = 5;
    for (int i = 0; i < 3; i++) // Matrix4 Population
    {
        for (int j = 0; j < 2; j++)
        {
            matrix4[i][j] = num;
            num -= 1;
        }
    }
    cout << matrix4 << endl;

    cout << "Identity matrix" << endl;
    Matrix identity = Matrix::identity(3);
    cout << identity << endl;

    cout << "Transpose of a 3x3 matrix (Matrix 1)" << endl;
    Matrix transpose1 = ~matrix1;
    cout << transpose1 << endl;

    cout << "Transpose of a 2x3 matrix (Matrix 3)" << endl;
    Matrix transpose2 = ~matrix3;
    cout << transpose2 << endl;

    cout << "Add same size matrix" << endl;
    Matrix matrixAdd(3, 3);
    matrixAdd = matrix1 + matrix2;
    cout << matrixAdd << endl;

    cout << "Add 2 different size matrices" << endl;
    try
```

```
79      {
80          matrixAdd = matrix1 + matrix3;
81      }
82      catch (...)
83      {
84          cout << "Caught adding 2 different sized matrices." << endl
85               << endl;
86      }
87
88      cout << "Multiply 2 same size matrices" << endl;
89      Matrix matrixMult3x3(3, 3);
90      matrixMult3x3 = matrix1 * matrix2;
91      cout << matrixMult3x3 << endl;
92
93      cout << "Multiply 2 different sized matrices properly (2x3)*(3x2)" << endl;
94      Matrix matrixMult2x2(2, 2); // matrix3 * matrix4
95      matrixMult2x2 = matrix3 * matrix4;
96      cout << matrixMult2x2 << endl;
97
98      cout << "Multiply 2 different sized matrices properly (3x2)*(2x3)" << endl;
99      matrixMult3x3 = matrix4 * matrix3;
100     cout << matrixMult3x3 << endl;
101
102     cout << "Multiply 2 different sized matrices improperly (3x3)*(2x3)" << endl;
103     try
104     {
105         matrixMult3x3 = matrix1 * matrix3;
106     }
107     catch (...)
108     {
109         cout << "Caught multiplying improper sized matrices" << endl
110              << endl;
111     }
112
113     cout << "Multiply scalar multiple. Matrix * scalar" << endl;
114     Matrix scalarMult1(3, 3);
115     scalarMult1 = matrix1 * 4;
116     cout << scalarMult1 << endl;
117
118     cout << "Multiply scalar multiple. Scalar * matrix" << endl;
119     Matrix scalarMult2(3, 3);
120     scalarMult2 = 4 * matrix1;
121     cout << scalarMult2 << endl;
122
123     cout << "Clear a matrix (Matrix 2)" << endl;
124     matrix2.clear();
125     cout << matrix2 << endl;
126
127     return 0;
128 }
```

```cpp
#include "matrix.h"
#include <iomanip>
using namespace std;

// constructor
Matrix::Matrix(int rows, int cols)
{
    if (rows <= 0 || cols <= 0)
    {
        throw std::out_of_range("The rows and columns must be greater than 0");
    }
    this->rows = rows;
    this->cols = cols;
    the_matrix = new Row *[rows];
    for (int i = 0; i < rows; i++)
    {
        the_matrix[i] = new Row(cols);
    }
}

// Copy constructor
Matrix::Matrix(const Matrix &from)
{
    rows = from.rows;
    cols = from.cols;
    the_matrix = new Row *[rows];
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            (*this)[i][j] = from[i][j];
        }
    }
}

// Destructor
Matrix::~Matrix()
{
    for (int i = 0; i < rows; i++)
    {
        delete the_matrix[i];
    }
    delete[] the_matrix;
}

// Assignment operator. Check row.cpp from Lab 2 to see more accurately how to do this.
Matrix &Matrix::operator=(const Matrix &rhs)
{
    rows = rhs.rows;
    cols = rhs.cols;
    for (int i = 0; i < rhs.rows; i++)
    {
        for (int j = 0; j < rhs.cols; j++)
        {
            (*this)[i][j] = rhs[i][j];
        }
    }
    return *this;
}

// Named Constructor
Matrix Matrix::identity(unsigned int size)
{
    Matrix result(size, size);
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i == j)
            {
                result[i][j] = 1;
            }
            else
            {
                result[i][j] = 0;
            }
        }
    }
```

```
79      return result;
80  }
81
82  // Matrix addition.
83  Matrix Matrix::operator+(const Matrix &rhs) const
84  {
85      // Check size is correct
86      if (rows != rhs.rows && cols != rhs.cols)
87      {
88          throw logic_error("Rows of both matrices and cols "
89                            "of both matrices must be equal");
90      }
91      Matrix result(rows, cols);
92      for (int i = 0; i < rows; i++)
93      {
94          for (int j = 0; j < cols; j++)
95          {
96              result[i][j] = (*this)[i][j] + rhs[i][j]; // not the_matrix[i][j]
97          }
98      }
99      return result;
100 }
101
102 // Matrix multiplication
103 Matrix Matrix::operator*(const Matrix &rhs) const
104 {
105     if (cols != rhs.rows)
106     {
107         throw logic_error("The cols of the first matrix "
108                           "must be equal to the rows of the second matrix.");
109     }
110     Matrix result(rows, rhs.cols);
111     for (int i = 0; i < result.rows; i++)
112     {
113         for (int j = 0; j < rhs.cols; j++)
114         {
115             for (int k = 0; k < cols; k++)
116             {
117                 result[i][j] += (*this)[i][k] * rhs[k][j];
118             }
119         }
120     }
121     return result;
122 }
123
124 // Scalar multiplication
125 Matrix Matrix::operator*(const double scale) const
126 {
127     Matrix result(this->rows, this->cols);
128     for (int i = 0; i < rows; i++)
129     {
130         for (int j = 0; j < cols; j++)
131         {
132             result[i][j] = ((*this)[i][j]) * scale;
133         }
134     }
135     return result;
136 }
137
138 // global scalar multiplication
139 Matrix operator*(const double scale, const Matrix &rhs)
140 {
141     Matrix result(rhs.rows, rhs.cols);
142     for (int i = 0; i < result.rows; i++)
143     {
144         for (int j = 0; j < result.cols; j++)
145         {
146             result[i][j] = scale * rhs[i][j];
147         }
148     }
149     return result;
150 }
151
152 // Transpose of a Matrix
153 Matrix Matrix::operator~() const
154 {
155     Matrix result(this->cols, this->rows);
156     for (int i = 0; i < this->rows; i++)
```

```
157     {
158         for (int j = 0; j < this->cols; j++)
159         {
160             result[j][i] = (*this)[i][j];
161         }
162     }
163     return result;
164 }
165
166 // Clear Matrix
167 void Matrix::clear()
168 {
169     for (int i = 0; i < rows; i++)
170     {
171         for (int j = 0; j < cols; j++)
172         {
173             (*this)[i][j] = 0;
174         }
175     }
176 }
177
178 // Access Operators - non-const
179 Row &Matrix::operator[](unsigned int row)
180 {
181     if (row < 0 || row >= rows)
182     {
183         throw out_of_range("Row cannot be less than 0 or "
184                            "greater than the amount of rows in matrix");
185     }
186     return *(the_matrix[row]);
187 }
188
189 // Access Operators - const
190 Row Matrix::operator[](unsigned int row) const
191 {
192     if (row < 0 || row >= rows)
193     {
194         throw out_of_range("Row cannot be less than 0 or "
195                            "greater than the amount of rows in matrix");
196     }
197     return *(the_matrix[row]);
198 }
199
200 // global insertion operator... ios_base
201 std::ostream &operator<<(std::ostream &os, const Matrix &rhs)
202 {
203     for (int i = 0; i < rhs.rows; i++)
204     {
205         cout << "[";
206         for (int j = 0; j < rhs.cols; j++)
207         {
208             os.precision(2);
209             os.width(3);
210             os << rhs[i][j];
211         }
212         os << "]" << endl;
213     }
214
215     return os;
216 }
```

```
#ifndef matrix_h
#define matrix_h

#include <iostream>
#include "row.h"
class Matrix
{
public:
    // No default (no argument) constructor.  It doesn't really make
    // sense to have one as we cannot rely on a size.  This may trip
    // us up later, but it will lead to a better implementation.

    // Constructor - create Matrix and clear cells.  If rows or
    // cols is < 1, throw an exception
    Matrix(int rows, int cols);

    // Copy constructor - make a new Matrix just like rhs
    Matrix(const Matrix &from);

    // Destructor.  Free allocated memory
    ~Matrix();

    // Assignment operator - make this just like rhs.  Must function
    // correctly even if rhs is a different size than this.
    Matrix &operator=(const Matrix &rhs);

    // Named Constructor - produce a square identity matrix of the
    // requested size.  Since we do not know how the object produced will
    // be used, we pretty much have to return by value.  A size of 0
    // would not make sense and should throw an exception.
    static Matrix identity(unsigned int size);

    // Matrix addition - lhs and rhs must be same size otherwise
    // an exception shall be thrown
    Matrix operator+(const Matrix &rhs) const;

    // Matrix multiplication - lhs and rhs must be compatible
    // otherwise an exception shall be thrown
    Matrix operator*(const Matrix &rhs) const;

    // Scalar multiplication.  Note, this function will support
    // someMatrixObject * 5.0, but not 5.0 * someMatrixObject.
    Matrix operator*(const double scale) const;

    // Matrix scalar multiplication when the scalar is first
    // 5.0 * someMatrixObject;
    friend Matrix operator*(const double scale, const Matrix &rhs);

    // Transpose of a Matrix - should always work, hence no exception
    Matrix operator~() const;

    // Clear Matrix to all members 0.0
    void clear();

    // Access Operators - throw an exception if index out of range
    Row &operator[](unsigned int row);

    // const version of above - throws an exception if indices are out of
    // range
    Row operator[](unsigned int row) const;

    friend std::ostream &operator<<(std::ostream &os, const Matrix &rhs);

private:
    // An array of Row pointers size "rows" that each point to a double array
    // of size "cols"
    Row **the_matrix;
    unsigned int rows;
    unsigned int cols;

    /** routines **/

    // add any "helper" routine here, such as routines to support
    // matrix inversion
};

/** Some Related Global Functions **/

```

```
79  // Overloaded global << with std::ostream as lhs, Matrix as rhs.  This method
80  // should generate output compatible with an ostream which is commonly used
81  // with console (cout) and files.  Something like:
82  // [[ r0c0, r0c1, r0c2 ]
83  //  [ r1c0, r1c1, r1c2 ]
84  //  [ r0c0, r0c1, r0c2 ]]
85  // would be appropriate.
86  //
87  // You should make this function a "friend" of the Matrix class so it can acess
88  // private data members
89  std::ostream &operator<<(std::ostream &os, const Matrix &rhs);
90
91  // We would normally have a corresponding >> operator, but
92  // will defer that exercise that until a later assignment.
93
94  // Scalar multiplication with a global function.  Note, this function will
95  // support 5.0 * someMatrixObject, but not someMatrixObject * 5.0
96  Matrix operator*(const double scale, const Matrix &rhs);
97
98  #endif
99  // Based on lab by Dr. Darrin Rothe ((c) 2015 Dr. Darrin Rothe)
```

Table of Contents