```cpp
/**
  ******************************************************************************
  * @file    : main.cpp
  * @brief   : Main Program
  *          : Lab 3: Matrix
  *          : CS-3210/021
  * @date    : MAR 30 2021
  * @author : Julian Singkham
  ******************************************************************************
  * @attention
  *
  * This program handles the creation of a matrix and the +,=, and * operators.
  *
  *
  ******************************************************************************
**/
#include<iostream>
#include "matrix.h"

using namespace std;
//=======================================Methods=======================================
 /**
  * @brief The program entry point. Assume tests are successful unles otherwise stated
  * by "Failed" or error is thrown.
  *
  * @param NOT USED
  *
  * @retval NOT USED
  */
int main(){
    // Test malformed matrix creation
    cout << "Test malformed matrix creation" << endl;
    try{
        matrix m1(0,-2);
        cout << "Failed" << endl;
    }
    catch(matrix_Exception& e){
        cout << e.what() << endl;
    }
    cout << endl;
    cout << endl;

    // Test proper matrix creation
    cout << "Test proper matrix creation" << endl;
    matrix m_out(2,2);
    try{
        cout << m_out << endl;
        if(m_out[0][0] != 0 && m_out[0][1] != 0 && m_out[1][0] != 0
           && m_out[0][0] != 0)
            cout << "Failed" << endl;
    }
    catch(matrix_Exception& e){
        cout << e.what() << endl;
    }
    cout << endl;
    cout << endl;

    // Test malformed matrix multiplication
    cout << "Test malformed matrix multiplication" << endl;
    matrix m1(2,2);
    m1[0][0] = 0;
    m1[0][1] = 1;
    m1[1][0] = 2;
    m1[1][1] = 3;
    cout << m1 << endl;

    matrix m2(3,2);
    m2[0][0] = 4;
    m2[0][1] = 5;
    m2[1][0] = 6;
    m2[1][1] = 7;
```

```
72      m2[2][0] = 8;
73      m2[2][1] = 9;
74      cout << m2 << endl;
75
76      m_out = matrix::identity(3);
77
78      try{
79          m_out = m1*m2;
80          cout << "Failed" << endl;
81      }
82      catch(matrix_Exception& e){
83          cout << e.what() << endl;
84      }
85      cout << endl;
86      cout << endl;
87
88      //Test proper matrix multiplication
89      cout << "Test proper matrix multiplication" << endl;
90      matrix m3(2,3);
91      m3[0][0] = 0;
92      m3[0][1] = 1;
93      m3[0][2] = 2;
94      m3[1][0] = 3;
95      m3[1][1] = 4;
96      m3[1][2] = 5;
97      cout << m3 << endl;
98
99      cout << m2 << endl;
100
101     try{
102         m_out = m3*m2;
103         cout << m_out << endl;
104         if(m_out[0][0] != 6 && m_out[0][1] != 7 && m_out[1][0] != 36
105            && m_out[0][0] != 43)
106             cout << "Failed" << endl;
107     }
108     catch(matrix_Exception& e){
109         cout << e.what() << endl;
110     }
111     cout << endl;
112     cout << endl;
113
114     //Test improper + operator
115     cout << "Test improper + operator" << endl;
116     cout << m1 << endl;
117     cout << m2 << endl;
118     cout << m3 << endl;
119     try{
120         m_out = m1 + m3;
121         cout << "Failed" << endl;
122     }
123     catch(matrix_Exception& e){
124         cout << e.what() << endl;
125     }
126     cout << endl;
127     cout << endl;
128
129     //Test proper + operator
130     cout << "Test proper + operator" << endl;
131     cout << m1 << endl;
132     m2 = matrix::identity(2);
133     cout << m2 << endl;
134
135     try{
136         m_out = m1 + m2;
137         cout << m_out << endl;
138
139         if(m_out[0][0] != 1 && m_out[0][1] != 1 && m_out[1][0] != 2
140            && m_out[0][0] != 4)
141             cout << "Failed" << endl;
142     }
```

```
143         catch(matrix_Exception& e){
144             cout << e.what() << endl;
145         }
146         cout << endl;
147         cout << endl;
148 
149         //Test scale multiplication
150         cout << "Test scale multiplication of 5" << endl;
151         cout << m1 << endl;
152 
153         m_out = m1 * 5;
154         cout << m_out << endl;
155         if(m_out[0][0] != 0 && m_out[0][1] != 5 && m_out[1][0] != 10
156            && m_out[0][0] != 15)
157             cout << "Failed" << endl;
158         cout << endl;
159         cout << endl;
160     }
```

```cpp
/**
  ******************************************************************************
  * @file    : matrixm.cpp
  * @brief   : matrix multiplier
  *          : Lab 2: Arrays and Matrices
  *          : CS-3210/021
  * @date    : MAR 23 2021
  * @author : Julian Singkham
  ******************************************************************************
  * @attention
  *
  * This API handles the creation, deletion, and =,+,* operators of a matrix as well
  * as retrieving matrix values using []. In essence the double matrix used in this
  * API is an array of arrays where the **double matrix points to rows *double arrays
  * that then point to col elements.
  *
  ******************************************************************************
**/
#include "matrix.h"
#include <string>
#include <cmath>

//=====================================Public=====================================
/**
 * @brief Makes the insertion operator a friend so it can access matrix.
 * Basically allows the matrix to be printed to std.
 *
 * @param os: Stream to write to.
 * @param rhs: Reference to the matrix that is being printed.
 *
 * @retval Stream containing the matrix printout.
 */
std::ostream& operator<<(std::ostream& os, const matrix& rhs){
    for(int i = 0; i < rhs.rows; i++){
        os << "|";
        for(int j = 0; j < rhs.cols; j++){
            double temp = rhs.the_matrix[i][j];
            os << temp << "|";
        }
        os << std::endl;
    }
    return os;
}

/**
 * @brief Matrix constructor, it creates a matrix of given dimensions
 * with clear (zeroed) cells. Throws error if dimensions are not possible (<1).
 *
 * @param rows: How many rows in the matrix.
 * @param cols: How many columns in the matrix.
 *
 * @retval NONE
 */
matrix::matrix(unsigned int rows, unsigned int cols) : rows(rows), cols(cols){
    if(rows < 1 || cols < 1)
        throw matrix_Exception("p-constructor has bad arguments.");

    the_matrix = new double*[rows];//Allocates memory to the # of rows

    //Allocate memory for each row of the array to the # of columns
    //Basically creates a 1-D array of 1-D arrays
    for(int i = 0; i < rows; i++)
        the_matrix[i] = new double[cols];

    clear(); //Fill matrix with zeroes
}

/**
 * @brief Copy constructor that makes a new matrix from a given one.
 *
 * @param from: matrix to copy into the new matrix.
```

```
72   *
73   * @retval A copy of the given matrix.
74   */
75  matrix::matrix(const matrix& from) : rows(from.rows), cols(from.cols){
76      the_matrix = new double*[rows];//Allocates memory to the # of rows
77
78      //Allocate memory for each row of the array to the # of columns
79      //Basically creates a 1-D array of 1-D arrays
80      for(int i = 0; i < rows; i++)
81          the_matrix[i] = new double[cols];
82
83      //Copy values from "from" into new matrix
84      for(int i = 0; i < rows; i++)
85          for(int j = 0; j < cols; j++)
86              the_matrix[i][j] = from[i][j];
87  }
88
89  /**
90   * @brief Frees allocated memory form matrix
91   *
92   * @param: NONE
93   *
94   * @retval NONE
95   */
96  matrix::~matrix(){
97      for(int i = 0; i <  rows; i++)
98          delete[] the_matrix[i]; //Delete each row of the matrix
99      delete[] the_matrix; //Delete the matrix itself
100 }
101
102 /**
103  * @brief Named constructor, it creates a square identity matrix
104  * of given size.
105  *
106  * Identity matrix is a matrix that is all zeros expect when
107  * row#=col#, then it is 1.
108  * // 1 0 0 [0][0] = 1
109  * // 0 1 0 [1][1] = 1
110  * // 0 0 1 [2][2] = 1
111  *
112  * @param size: Square dimensions of the matrix.
113  *
114  * @retval The square identity matrix.
115  */
116 matrix matrix::identity(unsigned int size){
117     if(size == 0)
118         throw matrix_Exception("Can not create an identity matrix of size 0.");
119
120     matrix return_matrix(size, size);
121     for(int i = 0; i < size; i++)
122         return_matrix[i][i] = 1.0;
123     return return_matrix;
124 }
125
126 /**
127  * @brief Assigns the matrix to the value stored in the given matrix.
128  *
129  * @param rhs: The given matrix to copy from.
130  *
131  * @retval A copy of the given matrix.
132  */
133 matrix& matrix::operator=(const matrix& rhs){
134     //Verify matrices match in size
135     if(rows != rhs.rows || cols != rhs.cols){
136         for(int i = 0; i <  rows; i++)
137             delete[] the_matrix[i]; //Delete each row of the matrix
138         delete[] the_matrix; //Delete the matrix itself
139
140         rows = rhs.rows;
141         cols = rhs.cols;
142         the_matrix = new double*[rows];//Allocates memory to the # of rows
```

```
143
144         //Allocate memory for each row of the array to the # of columns
145         //Basically creates a 1-D array of 1-D arrays
146         for(int i = 0; i < rows; i++)
147             the_matrix[i] = new double[cols];
148     }
149     //Copy values from rhs into current matrix
150     for(int i = 0; i < rows; i++)
151         for(int j = 0; j < cols; j++)
152             the_matrix[i][j] = rhs[i][j];
153
154     return *this;
155 }
156
157 /**
158  * @brief Matrix addiiton. The lhs and rhs must be the same size.
159  *
160  * @param rhs: The right hand side matrix.
161  *
162  * @retval The resulting matrix after addition.
163  */
164 matrix matrix::operator+(const matrix& rhs) const{
165     //Verify matrices match in size
166     if(rows != rhs.rows || cols != rhs.cols)
167         throw matrix_Exception("Size mismatch – The column/row of the left matrix does"
168                                " not match the column/row of the right matrix:");
169
170     matrix return_matrix(rows, cols);
171
172     for(int i = 0; i < rows; i++)
173         for(int j = 0; j < cols; j++)
174             return_matrix[i][j] = the_matrix[i][j] + rhs[i][j];
175
176     return return_matrix;
177 }
178
179 /**
180  * @brief Matrix multiplication.
181  * The lhs column size and rhs row size must match.
182  *
183  * @param rhs: The right hand side matrix.
184  *
185  * @retval The resulting matrix after multiplication.
186  * Dimension: lhs.rows x rhs.cols.
187  */
188 matrix matrix::operator*(const matrix& rhs) const{
189     //Verify matrices match in size
190     if(cols != rhs.rows)
191         throw matrix_Exception("Size mismatch – The column of the left matrix does"
192                                " not match the row of the right matrix:");
193
194     matrix return_matrix(rows, rhs.cols);
195
196     for (int i = 0; i < rows; ++i)
197         for (int j = 0; j < rhs.cols; ++j)
198             for (int k = 0; k < rhs.cols; ++k)
199                 return_matrix[i][j] += the_matrix[i][k] * rhs[k][j];
200     return return_matrix;
201 }
202
203 /**
204  * @brief Matrix scaler multiplication.
205  * This only supports matrix * 5, not 5 * matrix.
206  *
207  * @param scale: Value to scale the matrix.
208  *
209  * @retval The scaled matrix.
210  */
211 matrix matrix::operator*(const double scale) const{
212     matrix return_matrix(rows, cols);
213
```

```
214     for (int i = 0; i < rows; ++i)
215         for (int j = 0; j < cols; ++j)
216             return_matrix[i][j] = the_matrix[i][j] * scale;
217     return return_matrix;
218 }
219 
220 /**
221  * @brief This allows access of the matrix elements by using [].
222  *
223  * @param row: The desired row of the matrix.
224  *
225  * @retval A pointer to the desired element of the matrix.
226  */
227 double* matrix::operator[](unsigned int row){
228     //Verify row is within bounds
229     if (row >= rows || row < 0)
230         throw matrix_Exception("Size mismatch – The requested row is out of bounds.");
231 
232     double *ret = the_matrix[row];
233     return ret;
234 }
235 
236 /**
237  * @brief This allows access of the matrix elements by using [].
238  * Const version
239  *
240  * @param row: The desired row of the matrix.
241  *
242  * @retval A pointer to the desired element of the matrix.
243  */
244 double* matrix::operator[](unsigned int row) const{
245     //Verify row is within bounds
246     if (row >= rows || row < 0)
247         throw matrix_Exception("Size mismatch – The requested row is out of bounds.");
248 
249     double *ret = the_matrix[row];
250     return ret;
251 }
252 
253 /**
254  * @brief Zeroes the elements of the matrix.
255  *
256  * @param: None
257  *
258  * @retval None
259  */
260 void matrix::clear(){
261     for(int i = 0; i < rows; i++)
262         for(int j = 0; j < cols; j++)
263             the_matrix[i][j] = 0.0;
264     return;
265 }
266 
267 //=======================================Global=======================================
268 /**
269  * @brief Matrix scaler multiplication.
270  * This only supports matrix * 5, not 5 * matrix.
271  *
272  * @param scale: Value to scale the matrix.
273  * @param rhs: The matrix to apply the scaling.
274  *
275  * @retval The scaled matrix.
276  */
277 matrix operator*(const double scale, const matrix& rhs){
278     matrix return_matrix(rhs);
279     return_matrix = rhs * scale;
280     return return_matrix;
281 }
282 
```

```
CC = g++
CFLAGS = -c -MMD
LFLAGS = -Wall -Wextra -g
LDFLAGS ?= -lglut -lGLU  -lGL
SOURCES = $(wildcard *.cpp)
OBJECTS = $(SOURCES:.cpp=.o)
EXECUTABLE = ex

all: $(EXECUTABLE) clean
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LFLAGS) -o $@ $(OBJECTS) $(LDFLAGS)

%.o:%.cpp
    $(CC) $(CFLAGS) $<

-include *.d

clean:
    rm -f *.d
    rm -f *.o
```

**Table of Contents**