// matrix.cpp

// Homework 1

//

// zjc5164@psu.edu

//

// Created by Zhao Chen on 2/6/18.

// Copyright © 2018 Zhao Chen. All rights reserved.

//

#include <iostream>

#include <iomanip> // for std::setw

#include <stdlib.h> // for rand() function

#include <time.h> // time for srand

using namespace std;

class Matrix

{

public:

// non-default constructor

Matrix(int rows, int columns) : m\_rows(rows), m\_columns(columns)

{

m\_pArray = new int\*[m\_rows];

for(int m = 0; m < m\_rows; ++m)

{

m\_pArray[m] = new int[m\_columns];

}

}

// copy-constructor (allow returning Matrix from functions)

Matrix(const Matrix& matrix)

{

m\_rows = matrix.m\_rows;

m\_columns = matrix.m\_columns;

m\_pArray = new int\*[m\_rows];

for(int m = 0; m < m\_rows; ++m)

{

m\_pArray[m] = new int[m\_columns];

}

for (int m = 0; m < m\_rows; m++)

{

for (int n = 0; n < m\_columns; ++n)

{

m\_pArray[m][n] = matrix.m\_pArray[m][n];

}

}

}

// destructors

~Matrix()

{

for (int m = 0; m < m\_rows; ++m)

{

delete[] m\_pArray[m];

m\_pArray[m] = NULL;

}

delete[] m\_pArray;

m\_pArray = NULL;

}

// Matrix Addition

Matrix addMatrix(const Matrix &anotherMatrix)

{

Matrix newMatrix(m\_rows, m\_columns);

for( int m = 0; m < m\_rows; m++)

{

for( int n = 0; n < m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m\_pArray[m][n] + anotherMatrix.m\_pArray[m][n];

}

}

return newMatrix;

}

// Matrix Subtraction

Matrix substractMatrix(const Matrix &anotherMatrix)

{

Matrix newMatrix(m\_rows, m\_columns);

for( int m = 0; m < m\_rows; m++)

{

for( int n = 0; n < m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m\_pArray[m][n] - anotherMatrix.m\_pArray[m][n];

}

}

return newMatrix;

}

// Matrix multiplication scalar

Matrix multiply(int scalar)

{

Matrix newMatrix(m\_rows, m\_columns);

for (int m = 0; m < m\_rows; m++)

{

for (int n = 0; n < m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m\_pArray[m][n] \* scalar;

}

}

return newMatrix;

}

// Matrix multiplication

Matrix multiply(const Matrix &matrix2)

{

Matrix newMatrix(m\_rows, matrix2.m\_columns);

for (int m = 0; m < newMatrix.m\_rows; m++)

{

for (int n = 0; n < newMatrix.m\_columns; n++)

{

int sum = 0;

for (int i = 0; i < m\_columns; i++)

{

sum = sum + m\_pArray[m][i] \* matrix2.m\_pArray[i][n];

}

newMatrix.m\_pArray[m][n] = sum;

}

}

return newMatrix;

}

void print()

{

for (int m = 0; m < m\_rows; m++)

{

for (int n = 0; n < m\_columns; ++n)

{

cout << std::setw(3);

cout << m\_pArray[m][n] << " ";

}

cout << endl;

}

}

// Additional functions

int GetNumberOfRows() const

{

return m\_rows;

}

int GetNumberOfColumns() const

{

return m\_columns;

}

int \*\*m\_pArray; // Dynamically allocated 2d array

private:

int m\_rows;

int m\_columns;

};

// Assume input is correct

Matrix CreateMatrixFromInput(int matrixNumber)

{

cout << "Number of Rows in Matrix " << matrixNumber << ": ";

int rows, columns;

cin >> rows;

cout << "Number of Columns in Matrix " << matrixNumber << ": ";

cin >> columns;

cout << "Values of Matrix " << matrixNumber << " (expecting " << rows \* columns << "): ";

Matrix newMatrix(rows, columns);

for (int m = 0; m < rows; ++m)

{

for (int n = 0; n < columns; ++n)

{

cin >> newMatrix.m\_pArray[m][n];

}

}

return newMatrix;

}

int main()

{

Matrix matrix1(CreateMatrixFromInput(1));

Matrix matrix2(CreateMatrixFromInput(2));

cout << "Matrix Addition" << endl;

cout << "Matrix1 is: " << endl;

matrix1.print();

cout << "Matrix2 is: " << endl;

matrix2.print();

if (matrix1.GetNumberOfRows() == matrix2.GetNumberOfRows() && matrix1.GetNumberOfColumns() == matrix2.GetNumberOfColumns())

{

Matrix addMatrix = matrix1.addMatrix(matrix2);

cout << "Result is: " << endl;

addMatrix.print();

}

else

{

cout << "Skip Addition (Matrix Size Does not Match)" << endl;

}

cout << "Matrix Subtraction" << endl;

cout << "Matrix1 is: " << endl;

matrix1.print();

cout << "Matrix2 is: " << endl;

matrix2.print();

if (matrix1.GetNumberOfRows() == matrix2.GetNumberOfRows() && matrix1.GetNumberOfColumns() == matrix2.GetNumberOfColumns())

{

Matrix subMatrix = matrix1.substractMatrix(matrix2);

cout << "Result is: " << endl;

subMatrix.print();

}

else

{

cout << "Skip Subtraction (Matrix Size Does not Match)" << endl;

}

cout << "Matrix Multiplication Scalar" << endl;

cout << "Matrix1 is: " << endl;

matrix1.print();

srand( static\_cast<unsigned int>(time(NULL))); // initialize random

int iRandomNumber = rand() % 10 + 1;

cout << "Scalar value is: " << iRandomNumber << endl;

Matrix mulScalarMatrix = matrix1.multiply(iRandomNumber);

cout << "Result is: " << endl;

mulScalarMatrix.print();

cout << "Matrix Multiplication" << endl;

cout << "Matrix1 is: " << endl;

matrix1.print();

cout << "Matrix2 is: " << endl;

matrix2.print();

if (matrix1.GetNumberOfColumns() == matrix2.GetNumberOfRows())

{

Matrix mulMatrix = matrix1.multiply(matrix2);

cout << "Result is: " << endl;

mulMatrix.print();

}

else

{

cout << "Skip Multiplication (Matrix Size Does not Match)" << endl;

}

return 0;

}

// matrix\_ops.cpp

// Homework 1

//

// zjc5164@psu.edu

//

// Created by Zhao Chen on 2/6/18.

// Copyright © 2018 Zhao Chen. All rights reserved.

//

#include <iostream>

#include <iomanip> // for std::setw

#include <stdlib.h> // for rand() function

#include <time.h> // time for srand

using namespace std;

class Matrix

{

public:

// non-default constructor

Matrix(int rows, int columns) : m\_rows(rows), m\_columns(columns)

{

m\_pArray = new int\*[m\_rows];

for(int m = 0; m < m\_rows; ++m)

{

m\_pArray[m] = new int[m\_columns];

}

}

// copy-constructor (allow returning Matrix from functions)

Matrix(const Matrix& matrix)

{

m\_rows = matrix.m\_rows;

m\_columns = matrix.m\_columns;

m\_pArray = new int\*[m\_rows];

for(int m = 0; m < m\_rows; ++m)

{

m\_pArray[m] = new int[m\_columns];

}

for (int m = 0; m < m\_rows; m++)

{

for (int n = 0; n < m\_columns; ++n)

{

m\_pArray[m][n] = matrix.m\_pArray[m][n];

}

}

}

// destructors

~Matrix()

{

for (int m = 0; m < m\_rows; ++m)

{

delete[] m\_pArray[m];

m\_pArray[m] = NULL;

}

delete[] m\_pArray;

m\_pArray = NULL;

}

// Friend function

friend Matrix operator+(const Matrix &m1, const Matrix &m2);

friend Matrix operator-(const Matrix &m1, const Matrix &m2);

friend Matrix operator\*(const Matrix &m1, int scalar);

friend Matrix operator\*(int scalar, const Matrix &m1);

friend Matrix operator\*(const Matrix &m1, const Matrix &m2);

friend ostream& operator<<(ostream& out, const Matrix &m);

friend istream& operator>>(istream& input, Matrix &m);

// additional functions

int GetNumberOfRows() const

{

return m\_rows;

}

int GetNumberOfColumns() const

{

return m\_columns;

}

int \*\*m\_pArray; // dynamically allocated 2d array

private:

int m\_rows;

int m\_columns;

};

// Addition overloaded operator

Matrix operator+(const Matrix &m1, const Matrix &m2)

{

Matrix newMatrix(m1.m\_rows, m1.m\_columns);

for (int m = 0; m < newMatrix.m\_rows; m++)

{

for (int n = 0; n < newMatrix.m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m1.m\_pArray[m][n] + m2.m\_pArray[m][n];

}

}

return newMatrix;

}

// Multiplication scalar overloaded operator

Matrix operator\*(const Matrix &m1, int scalar)

{

Matrix newMatrix(m1.m\_rows, m1.m\_columns);

for (int m = 0; m < m1.m\_rows; m++)

{

for (int n = 0; n < m1.m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m1.m\_pArray[m][n] \* scalar;

}

}

return newMatrix;

}

// Subtraction overloaded operator

Matrix operator-(const Matrix &m1, const Matrix &m2)

{

Matrix newMatrix(m1.m\_rows, m1.m\_columns);

for( int m = 0; m < newMatrix.m\_rows; m++)

{

for( int n = 0; n < newMatrix.m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m1.m\_pArray[m][n] - m2.m\_pArray[m][n];

}

}

return newMatrix;

}

// Multiplication scalar overloaded operator

Matrix operator\*(int scalar, const Matrix &m1)

{

Matrix newMatrix(m1.m\_rows, m1.m\_columns);

for (int m = 0; m < m1.m\_rows; m++)

{

for (int n = 0; n < m1.m\_columns; ++n)

{

newMatrix.m\_pArray[m][n] = m1.m\_pArray[m][n] \* scalar;

}

}

return newMatrix;

}

// Multiplication overloaded opertor

Matrix operator\*(const Matrix &m1, const Matrix &m2)

{

Matrix newMatrix(m1.m\_rows, m2.m\_columns);

for (int m = 0; m < newMatrix.m\_rows; m++)

{

for (int n = 0; n < newMatrix.m\_columns; n++)

{

int sum = 0;

for (int i = 0; i < m1.m\_columns; i++)

{

sum = sum + m1.m\_pArray[m][i] \* m2.m\_pArray[i][n];

}

newMatrix.m\_pArray[m][n] = sum;

}

}

return newMatrix;

}

ostream& operator<<(ostream& out, const Matrix &matrix)

{

for (int m = 0; m < matrix.m\_rows; m++)

{

for (int n = 0; n < matrix.m\_columns; ++n)

{

out << std::setw(3);

out << matrix.m\_pArray[m][n] << " ";

}

out << endl;

}

return out;

}

istream& operator>>(istream& input, Matrix &matrix)

{

for (int m = 0; m < matrix.m\_rows; ++m)

{

for (int n = 0; n < matrix.m\_columns; ++n)

{

input >> matrix.m\_pArray[m][n];

}

}

return input;

}

// Assume input is correct

Matrix CreateMatrixFromInput(int matrixNumber)

{

cout << "Number of Rows in Matrix " << matrixNumber << ": ";

int rows, columns;

cin >> rows;

cout << "Number of Columns in Matrix " << matrixNumber << ": ";

cin >> columns;

cout << "Values of Matrix " << matrixNumber << " (expecting " << rows \* columns << "): ";

Matrix newMatrix(rows, columns);

std::cin >> newMatrix;

return newMatrix;

}

int main()

{

Matrix matrix1(CreateMatrixFromInput(1));

Matrix matrix2(CreateMatrixFromInput(2));

cout << "Matrix Addition" << endl;

cout << "Matrix1 is: " << endl << matrix1;

cout << "Matrix2 is: " << endl << matrix2;

if (matrix1.GetNumberOfRows() == matrix2.GetNumberOfRows() && matrix1.GetNumberOfColumns() == matrix2.GetNumberOfColumns())

{

Matrix addMatrix = matrix1 + matrix2;

cout << "Result is: " << endl << addMatrix;

}

else

{

cout << "Skip Addition (Matrix Size Does not Match)" << endl;

}

cout << "Matrix Subtraction" << endl;

cout << "Matrix1 is: " << endl << matrix1;

cout << "Matrix2 is: " << endl << matrix2;

if (matrix1.GetNumberOfRows() == matrix2.GetNumberOfRows() && matrix1.GetNumberOfColumns() == matrix2.GetNumberOfColumns())

{

Matrix subMatrix = matrix1 - matrix2;

cout << "Result is: " << endl << subMatrix;

}

else

{

cout << "Skip Subtraction (Matrix Size Does not Match)" << endl;

}

cout << "Matrix Multiplication Scalar" << endl;

cout << "Matrix1 is: " << endl << matrix1;

srand(static\_cast<unsigned int>(time(NULL))); // initialize random

int iRandomNumber = rand() % 10 + 1;

cout << "Scalar value is: " << iRandomNumber << endl;

Matrix mulScalarMatrix1 = matrix1 \* iRandomNumber;

cout << "Result is: " << endl << mulScalarMatrix1;

cout << "Matrix Multiplication Scalar" << endl;

cout << "Matrix1 is: " << endl << matrix1;

cout << "Scalar value is: " << iRandomNumber << endl;

Matrix mulScalarMatrix2 = iRandomNumber \* matrix1;

cout << "Result is: " << endl << mulScalarMatrix2;

cout << "Matrix Multiplication" << endl;

cout << "Matrix1 is: " << endl << matrix1;

cout << "Matrix2 is: " << endl << matrix2;

if (matrix1.GetNumberOfColumns() == matrix2.GetNumberOfRows())

{

Matrix mulMatrix = matrix1 \* matrix2;

cout << "Result is: " << endl << mulMatrix;

}

else

{

cout << "Skip Multiplication (Matrix Size Does not Match)" << endl;

}

return 0;

}