Matrix.h:

#include <fstream> // for file access

#include <iostream>

using namespace std;

#pragma once

class Matrix

{

private:

unsigned m\_rowSize; // cannot be negative, saves memory space

unsigned m\_colSize; // cannot be negative, saves memory space

double\*\* m\_matrix;

public:

Matrix();

Matrix(unsigned, unsigned, double); // holds row size, column size, initial value for each cell

Matrix(const Matrix&); // copy constructor

~Matrix(); // destructor

// Matrix Operations

Matrix operator+(Matrix&); // sum two matrices

Matrix operator-(Matrix&); // subtract two matrices

Matrix operator\*(Matrix&); // multiply two matrices

Matrix operator=(Matrix&);

Matrix transpose();

// Scalar Operations

Matrix operator+(double);

Matrix operator-(double);

Matrix operator\*(double);

Matrix operator/(double);

// Another Methods

double& operator()(const unsigned&, const unsigned&); // Matrix(3,4)

void print() const;

unsigned getRows() const;

unsigned getCols() const;

void getMatrixFromConsole() const;

void printPrimaryDiagonal(unsigned) const;

void printSecondaryDiagonal(unsigned) const;

void writeMatrixToFile() const;

};

Matrix.cpp:

#include "Matrix.h"

using namespace std;

// Default constructor

Matrix::Matrix()

{

m\_rowSize = 0;

m\_colSize = 0;

m\_matrix = NULL;

}

// Constructor for any matrix

Matrix::Matrix(unsigned rowSize, unsigned colSize, double initial = 0.0)

{

m\_rowSize = rowSize;

m\_colSize = colSize;

m\_matrix = new double\* [m\_rowSize];

for (unsigned row = 0; row < m\_rowSize; row++)

{

m\_matrix[row] = new double[m\_colSize];

for (unsigned col = 0; col < m\_colSize; col++)

{

m\_matrix[row][col] = initial;

}

}

}

// Copy constructor

Matrix::Matrix(const Matrix& other)

{

cout << "\nCopy constructor invoked\n";

m\_rowSize = other.m\_rowSize;

m\_colSize = other.m\_colSize;

m\_matrix = new double\* [other.m\_rowSize]; // create new instance

for (unsigned row = 0; row < m\_rowSize; row++)

{

m\_matrix[row] = new double[m\_colSize];

for (unsigned col = 0; col < m\_colSize; col++)

{

m\_matrix[row][col] = other.m\_matrix[row][col];

}

}

}

// Destructor

Matrix::~Matrix()

{

for (unsigned row = 0; row < m\_rowSize; row++)

{

delete m\_matrix[row];

}

delete[] m\_matrix;

}

// Addition of two matrices

Matrix Matrix::operator+(Matrix& other)

{

Matrix resultMatrix(m\_colSize, m\_rowSize, 0.0); // create new matrix instance

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

resultMatrix(row, col) = this->m\_matrix[row][col] + other(row, col);

}

}

return resultMatrix;

}

// Subtraction of two matrices

Matrix Matrix::operator-(Matrix& other)

{

Matrix resultMatrix(m\_colSize, m\_rowSize, 0.0);

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

resultMatrix(row, col) = this->m\_matrix[row][col] - other(row, col);

}

}

return resultMatrix;

}

// Multiplication of two matrices

Matrix Matrix::operator\*(Matrix& other)

{

Matrix resultMatrix(m\_rowSize, other.getCols(), 0.0);

if (m\_colSize == other.getRows())

{

double temp = 0.0;

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < other.getCols(); col++)

{

temp = 0.0;

for (unsigned k = 0; k < m\_colSize; k++)

{

temp += m\_matrix[row][k] \* other(k, col);

}

resultMatrix(row, col) = temp;

// cout << multiply(row,col) << " ";

}

// cout << endl;

}

return resultMatrix;

}

else

{

Matrix emptyMatrix(m\_rowSize, m\_colSize);

return emptyMatrix;

}

}

Matrix Matrix::operator=(Matrix& other)

{

swap(m\_matrix, other.m\_matrix);

swap(m\_rowSize, other.m\_rowSize);

swap(m\_colSize, other.m\_colSize);

return \*this;

}

// Scalar Addition

Matrix Matrix::operator+(double scalar)

{

Matrix result(m\_rowSize, m\_colSize, 0.0);

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

result(row, col) = this->m\_matrix[row][col] + scalar;

}

}

return result;

}

// Scalar Subtraction

Matrix Matrix::operator-(double scalar)

{

Matrix result(m\_rowSize, m\_colSize, 0.0);

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

result(row, col) = this->m\_matrix[row][col] - scalar;

}

}

return result;

}

// Scalar Multiplication

Matrix Matrix::operator\*(double scalar)

{

Matrix result(m\_rowSize, m\_colSize, 0.0);

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

result(row, col) = this->m\_matrix[row][col] \* scalar;

}

}

return result;

}

// Scalar Division

Matrix Matrix::operator/(double scalar)

{

Matrix result(m\_rowSize, m\_colSize, 0.0);

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

result(row, col) = this->m\_matrix[row][col] / scalar;

}

}

return result;

}

// Returns value of given location when asked in the form Matrix(x,y)

double& Matrix::operator()(const unsigned& rowNumber, const unsigned& colNumber)

{

return this->m\_matrix[rowNumber][colNumber];

}

// Row size getter

unsigned Matrix::getRows() const

{

return this->m\_rowSize;

}

// Col size getter

unsigned Matrix::getCols() const

{

return this->m\_colSize;

}

// Take any given matrices transpose and returns another matrix

Matrix Matrix::transpose()

{

Matrix transpose(m\_colSize, m\_rowSize, 0.0);

for (unsigned row = 0; row < m\_colSize; row++)

{

for (unsigned col = 0; col < m\_rowSize; col++)

{

transpose(row, col) = this->m\_matrix[col][row];

}

}

return transpose;

}

// Get matrix from console

void Matrix::getMatrixFromConsole() const

{

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

cin >> m\_matrix[row][col];

}

}

}

void Matrix::printPrimaryDiagonal(unsigned m\_rowSize) const

{

cout << "Primary diagonal: ";

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_rowSize; col++)

{

if (row == col)

{

cout << m\_matrix[row][col] << ", ";

}

}

}

cout << endl;

}

void Matrix::printSecondaryDiagonal(unsigned m\_rowSize) const

{

cout << "Secondary diagonal: ";

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_rowSize; col++)

{

if ((row + col) == (m\_rowSize - 1))

{

cout << m\_matrix[row][col] << ", ";

}

}

}

cout << endl;

}

void Matrix::writeMatrixToFile() const

{

ofstream file("matrix.txt");

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

file << "[" << m\_matrix[row][col] << "]";

}

file << "\n";

}

file.close();

}

void Matrix::print() const

{

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

cout << "[" << m\_matrix[row][col] << "]";

}

cout << endl;

}

}

Main.cpp:

#include <fstream> // for file access

#include <iostream>

#include "Matrix.h"

using namespace std;

void readMatrixFromFile(Matrix& matrix, unsigned m\_rowSize, unsigned m\_colSize);

int main()

{

int m\_rowSize, m\_colSize;

cout << "Enter rows and columns of the matrix:\n";

cin >> m\_rowSize >> m\_colSize;

Matrix firstMatrix(m\_rowSize, m\_colSize, 0.0);

cout << "\nEnter the matrix elements one by one:\n";

firstMatrix.getMatrixFromConsole();

cout << "\nEntered matrix is:\n";

firstMatrix.print();

Matrix secondMatrix = firstMatrix; // invoke copy constructor

cout << "\nResult of the copy constructor is:\n";

secondMatrix.print();

Matrix duplicatedMatrix;

duplicatedMatrix = firstMatrix; // invoke operator=

cout << "\nResult of assignment operator:\n";

duplicatedMatrix.print();

Matrix transposedMatrix = duplicatedMatrix.transpose();

cout << "\nResult of transposed matrix:\n";

transposedMatrix.print();

cout << "\nResult of new matrix with initial values:\n";

Matrix testMatrix(m\_rowSize, m\_colSize, 7);

testMatrix.print();

cout << "\nPrint primary and secondary diagonals:\n";

transposedMatrix.printPrimaryDiagonal(m\_rowSize);

transposedMatrix.printSecondaryDiagonal(m\_rowSize);

cout << "\nRead matrix from file:\n";

Matrix matrix(m\_rowSize, m\_colSize, 0.0);

readMatrixFromFile(matrix, m\_rowSize, m\_colSize);

matrix.print();

// Addition of two matrices

Matrix additionMatrix = matrix + transposedMatrix;

cout << "\nAddition of matrix from file with transposed matrix:\n";

additionMatrix.print();

// Subtraction of two matrices

Matrix subtractionMatrix = matrix - transposedMatrix;

cout << "\nSubtraction of matrix from file with transposed matrix:\n";

subtractionMatrix.print();

// Multiplication of two matrices

Matrix multiplicationMatrix = matrix \* transposedMatrix;

cout << "\Multiplication of matrix from file with transposed matrix:\n";

multiplicationMatrix.print();

// Scalar Addition

Matrix matrixWithAddedScalar = testMatrix + 3;

matrixWithAddedScalar.print();

}

// External method to read matrix from file

void readMatrixFromFile(Matrix& matrix, unsigned m\_rowSize, unsigned m\_colSize)

{

ifstream file("matrix.txt");

for (unsigned row = 0; row < m\_rowSize; row++)

{

for (unsigned col = 0; col < m\_colSize; col++)

{

file >> matrix(row, col);

}

}

file.close();

}