#pragma once

#include "matrix.h"

template<typename T>matrix<T>::matrix(const matrix<T>& mtrx)

{

(\*this).ptr = nullptr;

(\*this) = mtrx;

}

template<typename T>matrix<T>::matrix()

{

this->colsize = 0;

this->rowsize = 0;

ptr = nullptr;

}

template<typename T>matrix<T>::matrix(uint64\_t colsize, uint64\_t rowsize)

{

this->colsize = colsize;

this->rowsize = rowsize;

ptr = new T[colsize \* rowsize];

for (uint64\_t i = 0; i < colsize \* rowsize; i++)

{

ptr[i] = 0;

}

}

template<typename T>matrix<T>::~matrix()

{

delete[] ptr;

}

template<typename T>matrix<T> matrix<T>::sqprediag(const uint64\_t S)const

{

matrix<T> temp(S, S);

for (int i = S - 1; i >= 0; i--)

{

temp[i][S - 1 - i] = 1;

}

return temp;

}

template<typename T>matrix<T> matrix<T>::transpose() const {

matrix<T> temp(rowsize, colsize);

for (size\_t i = 0; i < colsize; ++i) {

for (size\_t j = 0; j < rowsize; ++j) {

temp[j][i] = (\*this)[i][j];

}

}

return temp;

}

template<typename T>matrix<T> matrix<T>::inverse\_M()const

{

if (colsize != rowsize)

{

throw std::invalid\_argument("the number of rows in the matrix is not equal to the number of columns");

}

uint64\_t sizeM = (\*this).colsize;

matrix<T> a = (\*this), reversM, E\_lower\_diagonal, t1;

E\_lower\_diagonal = a.sqprediag(sizeM);

reversM = E\_lower\_diagonal \* E\_lower\_diagonal;

t1 = a.to\_uptrng(reversM);

t1 = ((t1 \* E\_lower\_diagonal).transpose() \* E\_lower\_diagonal).transpose();

reversM = ((reversM \* E\_lower\_diagonal).transpose() \* E\_lower\_diagonal).transpose();

t1 = t1.to\_uptrng(reversM);

t1 = ((t1 \* E\_lower\_diagonal).transpose() \* E\_lower\_diagonal).transpose();

reversM = ((reversM \* E\_lower\_diagonal).transpose() \* E\_lower\_diagonal).transpose();

for (uint64\_t i = 0; i < sizeM; i++)

{

T koef = t1[i][i];

if (koef == 0)

{

throw std::invalid\_argument("the matrix is not reversible");

}

for (uint64\_t k = 0; k < sizeM; k++)

{

t1[i][k] = t1[i][k] / koef;

reversM[i][k] = reversM[i][k] / koef;

}

}

return reversM;

}

template<typename T>matrix<T> matrix<T>::to\_uptrng(matrix<T>& other)const

{

if ((this->colsize == this->rowsize) && (this->colsize == (other.colsize)) && ((other.colsize) == other.rowsize))

{

matrix<T> temp((\*this));

for (uint64\_t i = 0; i < rowsize - 1; i++) {

for (uint64\_t j = i + 1; j < rowsize; j++) {

while (temp[i][i] == 0 && temp[j][i] == 0) {

j++;

if (j >= rowsize) {

throw std::invalid\_argument("the matrix is irreducible to the triangular form");

}

}

if (temp[i][i] == 0) {

for (uint64\_t k = 0; k < rowsize; k++) {

std::swap(temp[i][k], temp[j][k]);

std::swap(other[i][k], other[j][k]);

}

}

T koef1 = temp[j][i] / temp[i][i];

for (uint64\_t k = 0; k < rowsize; k++) {

temp[j][k] = temp[j][k] - temp[i][k] \* koef1;

other[j][k] = other[j][k] - other[i][k] \* koef1;

}

}

}

return temp;

}

throw std::invalid\_argument("[ ((this->colsize == this->rowsize) && (this->colsize == (other.colsize)) && ((other.colsize) == other.rowsize)) ]==0");

}

template<typename T>matrix<T> matrix<T>::to\_uptrng()const

{

matrix<T> temp((\*this));

T det;

det = 1;

for (uint64\_t i = 0; i < rowsize - 1; i++) {

for (uint64\_t j = i + 1; j < rowsize; j++) {

while (temp[i][i] == 0 && temp[j][i] == 0) {

j++;

if (j >= rowsize) {

throw std::invalid\_argument("the matrix is irreducible to the triangular form");

}

}

if (temp[i][i] == 0) {

for (uint64\_t k = 0; k < rowsize; k++) {

det = det \* (-1);

}

}

T koef = temp[j][i] / temp[i][i];

for (uint64\_t k = i; k < rowsize; k++) {

temp[j][k] = temp[j][k] - temp[i][k] \* koef;

}

}

}

for (uint64\_t i = 0; i < rowsize - 1; i++) {

temp[0][i] = (temp[0][i]) \* det;

}

return temp;

}

template <typename T>T matrix<T>::determinant() const {

if (rowsize != colsize) {

throw std::invalid\_argument("rowsize != colsize");

}

matrix<T> temp((\*this).to\_uptrng());

T det;

det = (T)(1);

std::cout <<"temp:\n" << temp << "\n";

for (uint64\_t i = 0; i < rowsize; i++) {

det = (det \*temp[i][i]);

}

return det;

}

template<typename T> std::ostream& operator<<(std::ostream& out, const matrix<T>& mtrx)

{

out << "sizex:" << mtrx.getcol() << "sizey:" << mtrx.getrow() << "\n";

for (uint64\_t i = 0; i < mtrx.getcol(); i++)

{

for (uint64\_t j = 0; j < mtrx.getrow(); j++)

{

out << "[" << i << "][" << j << "] = \t" << mtrx[i][j] << "\t | ";

}

out << "\n";

}

return out;

}

template<typename T>std::istream& operator>>(std::istream& in, matrix<T>& mtrx)

{

uint64\_t size;

std::cout << "Enter number of rows: ";

in >> size;

mtrx.setrow(size);

std::cout << "Enter number of columns: ";

in >> size;

mtrx.setcol(size);

mtrx.allocateMemory();

for (uint64\_t i = 0; i < mtrx.getrow(); i++) {

for (uint64\_t j = 0; j < mtrx.getcol(); j++) {

std::cout << "["<<i<<"]["<<j<<"]=";

in >> mtrx[i][j];

}

}

return in;

}

template<typename T>matrix<T>& matrix<T>::operator=(const matrix<T>& other)

{

if (this != &other)

{

delete[] ptr;

ptr = new T[other.getcol() \* other.getrow()];

this->colsize = other.getcol();

this->rowsize = other.getrow();

for (uint64\_t i = 0; i < other.getrow(); i++)

{

for (uint64\_t j = 0; j < other.getcol(); j++)

{

(\*this)[i][j] = other[i][j];

}

}

}

return \*this;

}

template<typename T>matrix<T> matrix<T>::operator-() const

{

matrix<T> result(this->getcol(), this->getrow());

for (uint64\_t i = 0; i < this->getrow(); i++)

{

for (uint64\_t j = 0; j < this->getcol(); j++)

{

result[i][j] = (-1)\*((\*this)[i][j]);

}

}

return result;

}

template<typename T>matrix<T> matrix<T>::operator+(const matrix<T>& other) const

{

matrix<T> result(other.getcol(), other.getrow());

if ((other.getcol()) == ((\*this).getcol()) && (other.getrow() == (\*this).getrow())) {

for (uint64\_t i = 0; i < other.getrow(); i++)

{

for (uint64\_t j = 0; j < other.getcol(); j++)

{

result[i][j] = (other[i][j] + (\*this)[i][j]);

}

}

}

else {

throw std::invalid\_argument("((other.getcol()) == ((\*this).getcol()) && (other.getrow() == (\*this).getrow())) == false");

}

return result;

}

template<typename T>matrix<T> matrix<T>::operator-(const matrix<T>& other) const

{

matrix<T> result(other.getcol(), other.getrow());

if ((other.getcol()) == ((\*this).getcol()) && (other.getrow() == (\*this).getrow())) {

for (uint64\_t i = 0; i < other.getrow(); i++)

{

for (uint64\_t j = 0; j < other.getcol(); j++)

{

result[i][j] = (other[i][j] - (\*this)[i][j]);

}

}

}

return result;

}

template<typename T>matrix<T> matrix<T>::operator\*(const T& other) const

{

matrix<T> result(this->getcol(), this->getrow());

for (uint64\_t i = 0; i < this->getrow(); i++)

{

for (uint64\_t j = 0; j < this->getcol(); j++)

{

result[i][j] = other \* ((\*this)[i][j]);

}

}

return result;

}

template<typename T>matrix<T> matrix<T>::operator\*(const matrix<T>& other) const

{

if (rowsize != other.colsize)

{

throw std::invalid\_argument("Matrix dimensions are not compatible for multiplication.");

}

matrix<T> result(colsize, other.rowsize);

for (uint64\_t i = 0; i < colsize; ++i) {

for (uint64\_t j = 0; j < other.rowsize; ++j) {

for (uint64\_t k = 0; k < rowsize; ++k) {

result[i][j] = result[i][j] + ((\*this)[i][k] \* other[k][j]);

}

}

}

return result;

}

template<typename T>matrix<T> matrix<T>::operator/(const matrix<T>& other) const

{

if (other.determinant() != 0) {

return (\*this) \* (other.inverse\_M());

}

throw std::invalid\_argument("the matrix on the right is not reversible");

}

template<typename T>void matrix<T>::allocateMemory()

{

delete[] ptr;

ptr = new T[colsize \* rowsize];

for (uint64\_t i = 0; i < colsize \* rowsize; i++)

{

ptr[i] = 0;

}

}