#pragma once

#include<iostream>

#include <sstream>

// //

// logic of the apply Method To Elements method using SFINAE to check for the presence of a

// method with a parameter for a matrix element and to call it if it exists

//the main template/

template <typename T, typename Param, typename = void>

struct HasMethodWithParam : std::false\_type {};

//Now the compiler is thinking whether to use the main template, or somewhere there is a separate specialization for such a case.

template <typename T, typename Param>

struct HasMethodWithParam<T, Param, std::void\_t<decltype(std::declval<T>().output\_mode\_set(std::declval<Param>()))>> : std::true\_type {};

// //

template<typename T> class matrix;

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

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

template <typename T> class matrix

{

private:

//an array with T elements

T\* ptr;

uint64\_t colsize;

uint64\_t rowsize;

//re-allocation of memory(does not save old values)

void allocateMemory();

public:

//CONSTRUCTORS\DESTRUCTORS

//the copying constructor

matrix(const matrix<T>& mtrx);

//square matrix constructor

matrix(uint64\_t size\_diag) :matrix(size\_diag, size\_diag) {}

//constructor with dimensions

matrix(uint64\_t colsize, uint64\_t rowsize);

//the default constructor

matrix();

~matrix();//destructor

//DATA ACCESS

uint64\_t getcol()const { return colsize; }

uint64\_t getrow()const { return rowsize; }

void setcol(uint64\_t colsize) { this->colsize = colsize; this->allocateMemory(); }

void setrow(uint64\_t rowsize) { this->rowsize = rowsize; this->allocateMemory(); }

//to index1 row access operator

T\* operator[](const uint64\_t index1) const { return ptr + index1 \* rowsize; }

//ARITHMETIC OPERATORS

// the unary operator returns a matrix with inverse (multiplied by minus 1 ) elements

matrix<T> operator-() const;

// binary matrix addition

matrix<T> operator+(const matrix<T>& other) const;

// binary matrix subtraction

matrix<T> operator-(const matrix<T>& other) const;

// binary matrix multiplication

matrix<T> operator\*(const matrix<T>& other) const;

// binary matrix division(if not a singular matrix on the left)

matrix<T> operator/(const matrix<T>& other) const;

// binary matrix multiplication by an element from the field

matrix<T> operator\*(const T& other) const;

// assignment operator overload

matrix<T>& operator=(const matrix<T>& other);

// I/O OPERATIONS

// overloading the output operator

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

// overloading the input operator

template<typename T>friend std::istream& operator>><>(std::istream& in, matrix<T>& p);

//SPECIAL METHODS

//return of the upper triangular matrix after transformations

matrix to\_uptrng()const;

//bringing to the upper triangular view together with the "other" matrix

matrix to\_uptrng(matrix<T>& other)const;

//matrix transposition

matrix transpose()const;

// finding the determinant if there is one

T determinant() const;

//return of the square matrix from 1 to the lower diagonal

// <-----S---->

// 0 0 .. 0 1 |

// 0 0 .. 1 0 |

// . .. ... .. .. S

// 0 1 .. 0 0 |

// 1 0 .. 0 0 |

//

matrix sqprediag(const uint64\_t S)const;

//return of the inverse matrix

matrix inverse\_M()const;

//applyMethodToElements function using SFINAE to check for a method with a parameter and call it

//for example:

//Matrix<ExampleClass> matrix;

//matrix.applyMethodToElements(&ExampleClass::output\_mode\_set, 2);

//a method that applies a method( output\_mode\_set(M) ) of class T to each element of the matrix

//(Not tested. Everything worked in the simplified code)

template <typename T, typename ReturnType, typename Param>

std::enable\_if\_t<HasMethodWithParam<T, Param>::value> applyMethodToElements(ReturnType(T::\* method)(Param), Param param)const////the implementation is not displayed in the cpp file((

{

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

(ptr[i].\*method)(param);

}

}

};

#include "matrix.cpp"