**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчёт по лабораторной работе

**Реализация классов для работы с матрицами и векторами**

Выполнил:

студент ИИТММ гр. 381906-2

Шелепин Н. А

Проверил:

ассистент каф. математического обеспечения и суперкомпьютерных технологий

Лебедев И.Г.

Нижний Новгород

2020 г.

Содержание

[Введение 4](#_Toc40884517)

[1. Постановка задачи 5](#_Toc40884518)

[2. Руководство пользователя 6](#_Toc40884519)

[3. Руководство программиста 8](#_Toc40884520)

[3.1. Описание структуры программы 8](#_Toc40884521)

[3.2. Описание структур данных 8](#_Toc40884529)

[3.3. Описание алгоритмов 11](#_Toc40884538)

[4. Эксперименты 12](#_Toc40884539)

[Заключение 16](#_Toc40884540)

[Литература 17](#_Toc40884541)

[Приложение 18](#_Toc40884542)

# Введение

Одними из важнейших понятий математической части современного программирования являются векторы и матрицы.

Матрица — математический объект, представляющий из себя прямоугольную таблицу чисел и допускающий алгебраические операции (сложение, вычитание, умножение) между ним и другими подобными объектами. Обычно матрицы представляются двумерными (прямоугольными) таблицами. Иногда рассматривают многомерные матрицы или матрицы непрямоугольной формы.

Объект, состоящий из одного столбца или строки, заполненной числами, называется вектором. Для векторов тоже определены операции сложения, умножения и т.д.

Работу с матрицами и векторами в программировании можно реализовать с помощью классов.

Класс – это тип структуры, позволяющий включать в описание типа не только переменные, но и функции (функции-элементы или методы).

В этой работе нам потребуются использовать шаблоны функций и классов. Шаблоны позволяют решить проблему необходимости написания одних и тех же алгоритмов несколько раз для различных типов.

Существуют шаблоны функций и шаблоны классов.

Шаблоны функций – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов. Другими словами, шаблон функции (шаблонная функция, обобщённая функция) представляет собой семейство разных функций (или описание алгоритма). По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными.

Шаблоны классов – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов.

Данная работа демонстрирует реализацию классов для работы с матрицами и векторами с использованием шаблонов.

# Постановка задачи

1. Реализовать классы для работы с векторами и матрицами, использовать шаблоны.
2. Реализовать сортировки: пузырьком, вставкой, быстрая;
3. Продемонстрировать их работу на примере (написать в main пример);
4. Должны быть:

* конструкторы (по умолчанию, инициализатор, копирования);
* деструктор;
* доступ к защищенным полям;
* перегруженные операции: +, -, \*, /, =, ==, [];
* потоковый ввод и вывод;
* перегруженные операции +, -, \*, / должны быть реализованы для векторов (вектор +, -, \*, / вектор), матриц (матрица +, -, \* матрица), матрично-векторные (матрица \* вектор и наоборот);
* в классе вектор должна быть возможность отсортировать его тремя способами.
* метод по нахождению конкретного элемента и метод нахождения всех вхождений конкретного элемента.
* метод деления матриц

1. Сравнить время работы и сделать вывод.

# Руководство пользователя

Использование программы пользователем:

1. Создать необходимы матрицы и векторы определенных типов.
2. Заполнить матрицы и векторы числами.
3. Произвести необходимые арифметические операции.
4. Записать результаты времени работы алгоритмов.

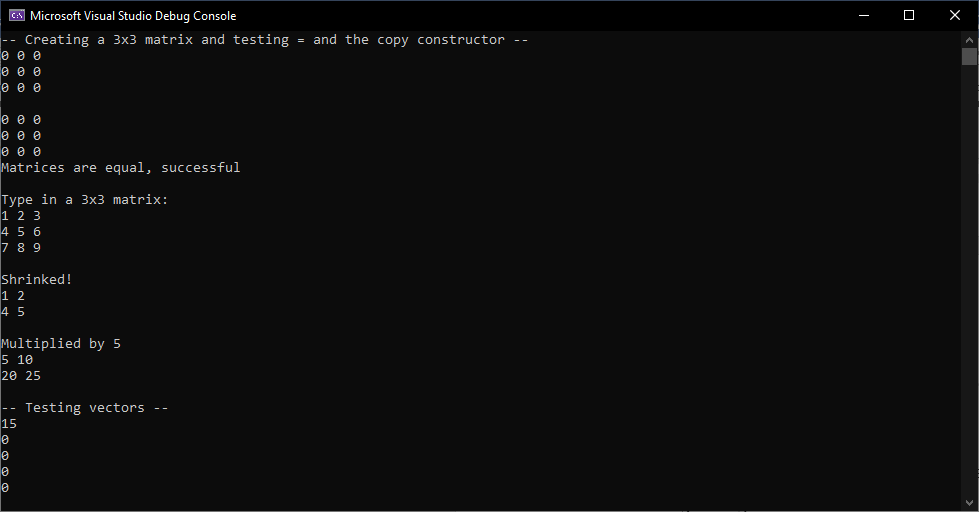


Рисунок 1. Cкриншот интерфейса программы.

# 

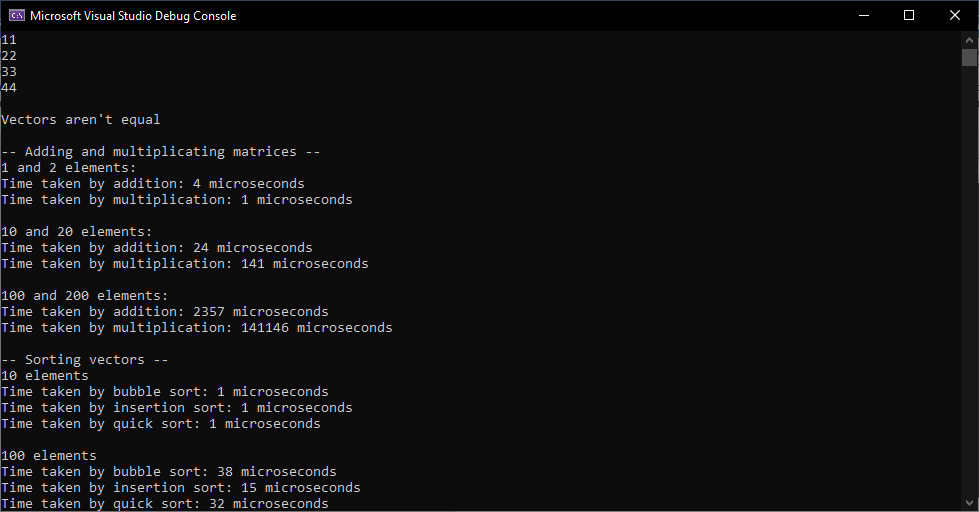


Рисунок 2. Cкриншот интерфейса программы.

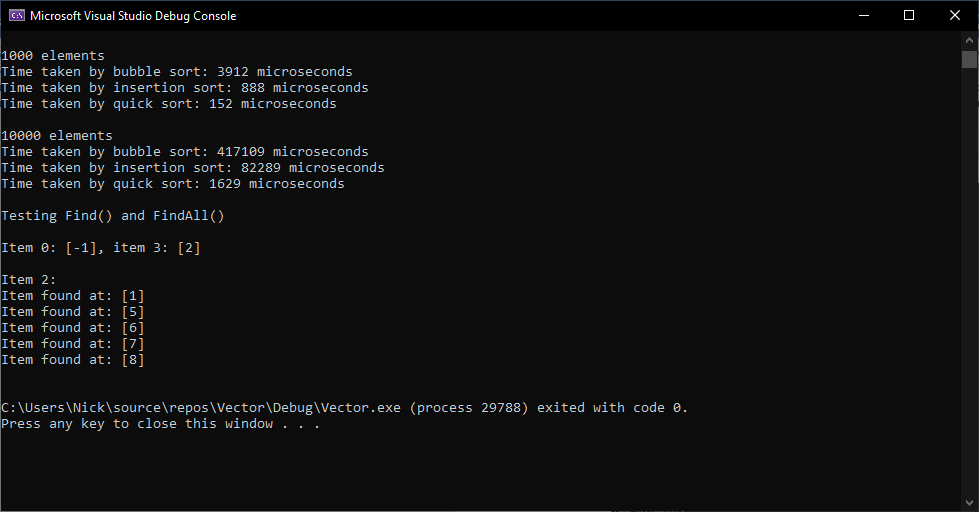


Рисунок 3. Cкриншот интерфейса программы.

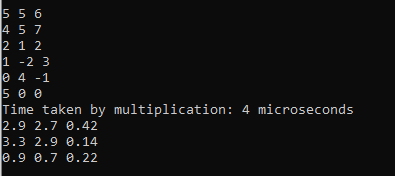


Рисунок 4. Cкриншот работы деления матриц.

# 3. Руководство программиста

## 3.1. Описание структуры программы

## Программа состоит из одного решения, называющегося Vector.

## В решении Vector содержится 5 модулей: Matrix.h, Matrix\_imp.h, Vector.h, Vector\_imp.h, main.cpp.

## В модуле Matrix.h определен класс TMatrix, а также объявлены все его методы;

## В модуль Matrix\_imp.h вынесены все реализации методов класса TMatrix;

## В модуле Vector.h определен класс TVector, являющийся наследником класса TMatrix;

## В модуль Vector\_imp.h вынесены все определения методов класса TVector;

## В модуле main.cpp определена стандартная функция int main(), внутри которой содержится набор действий с объектами типа TMatrix и TVector.

## 3.2. Описание структур данных

## В программе определены следующие шаблонные классы: Class TMatrix и Class TVector.

## Внутри класса TMatrix определены следующие protected-поля и методы:

## T\*\* matrix – шаблонный двойной указатель, двухмерный массив чисел матрицы;

## int rows – количество строк матрицы;

## int columns –количество столбцов матрицы;

* void inversion() – метод нахождения обратной матрицы (работает только с квадратными и невырожденными матрицами!!!)

## Внутри класса TMatrix определен следующий набор public-методов

## (плюс конструкторы и деструктор):

* TMatrix() – конструктор по умолчанию, не принимает никаких параметров, инициализирует все три поля 0 через списки инициализации;
* TMatrix(int \_r, int \_c) – конструктор инциализатор, принимает на вход два параметра типа int, создает динамический массив заданных размеров, заполняет его 0, поля rows и columns приравниваются \_r и \_c;
* TMatrix(const TMatrix& rhs) – конструктор копирования, принимает на вход объект типа TMatrix, создает объект с теми же характеристиками, что и переданный, и матрицей того же содержимого;
* TMatrix(int \_r, int \_c, int a, int b) – конструктор, принимающий на вход размеры матрицы и минимальное и максимальное значение для заполнения матрицы случайными числами.
* Virtual ~TMatrix() – виртуальный деструктор, очищает выделенную методами и конструкторами память;
* int get\_rows () const – метод, возвращающий количество строк матрицы;
* int get\_ columns() const – метод, возвращающий количество столбцов матрицы;
* T\*\* get\_array() const – метод, возвращающий массив, содержащий в себе все элементы матрицы;
* void set\_size(int new\_r, int new\_c) – метод, принимающий новые размеры матрицы, приводящий матрицу к этим размерам. В случае если новая матрица больше старой, новые элементы равняются нулю. В противном случае, все элементы, на которые не хватило места, «отрезаются»;
* TMatrix<T>& operator =(const TMatrix<T> & rhs) - перегрузка оператора =, позволяет присваивать один объект типа Matrix другому;
* T\*& operator [](int) - перегрузка оператора индексации, позволяет возвращать элемент матрицы по индексу;
* friend TMatrix<P> operator +(TMatrix<P>& lhs, TMatrix<P>& rhs) – перегрузка оператора суммы, позволяет возвращать матрицу, являющуюся суммой двух матриц;
* friend TMatrix<M> operator -(TMatrix<M>& lhs, TMatrix<M>& rhs) – перегрузка оператора ­– , аналогична в своей сути перегрузке оператора +;
* friend bool operator ==(TMatrix<E>& lhs, TMatrix<E>& rhs) – перегрузка оператора сравнения, позволяет сравнивать объекты типа TMatrix;
* friend TMatrix<M> operator \*(TMatrix<M>& lhs, TMatrix<M>& rhs) – перегрузка оператора \*, умножение матрицы на матрицу;
* friend TMatrix<M> operator \*(TMatrix<M>& lhs, N rhs) – перегрузка оператора \*, умножение матрицы на число;
* friend TMatrix<D> operator /(TMatrix<D>& lhs, N rhs) – перегруза оператора /, деление матрицы на число;
* friend ostream& operator <<(ostream& out, const TMatrix& rhs) – перегрузка оператора вывода, позволяет выводить матрицы на экран;
* friend istream& operator >>(istream& in, TMatrix& rhs) - перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера либо задавать размер и вводить самостоятельно.
* int Find(T a) – метод поиска вхождения элемента в матрицу, возвращает первый индекс вхождения;
* void FindAll(T a) – метод поиска всех вхождений элемента в матрицу.
* friend TMatrix<D> operator /(TMatrix<D>& lhs, TMatrix<D>& rhs) – перегрузка оператора / для деления матриц по методу умножения матрицы А на обратную к матрице B.

## Шаблонный класс TVector является public, наследником класса TMatrix, поэтому никаких полей в этом классе не определено. Внутри него, определены только специфичные для него методы:

* const TVector<T>& operator=(const std::initializer\_list<T>& list) – перегрузка оператора =, заполнение вектора с помощью списка инициализации;
* const TVector<T>& operator =(const TMatrix<T>& rhs) – перегрузка оператора = для присвоения вектору матрицы;
* friend TVector<N> operator +(TVector<P>& lhs, TVector<N>& rhs) – перегрузка +, сложение векторов;
* friend TVector<N> operator -(TVector<M>& lhs, TVector<N>& rhs) – перегрузка -, вычитание векторов;
* friend TVector<D> operator /(TVector<D>& lhs, N rhs) – перегрузка /, деление вектора на число;
* friend TVector<M> operator \*(TMatrix<M>& lhs, TVector<N>& rhs) – перегрузка \*, умножение матрицы на вектор;
* friend TMatrix<M> operator \*(TVector<M>& lhs, TMatrix<N>& rhs) – перегрузка \*, умножение вектора на матрицу;
* T\* get\_vector() – метод, возвращающий массив, содержащий в себе все элементы объекта Vector;
* T& operator [](const int i) const – перегрузка оператора индексации, позволяет возвращать элемент вектора по индексу;
* friend istream& operator >>(istream& in, TVector<I>& t) – перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера либо задавать размер и вводить самостоятельно;
* friend ostream& operator <<(ostream& out, const TVector<O>& t) – перегрузка оператора вывода;
* void BubbleSort() - сортировка методом «Пузырек»;
* void InsertionSort() - сортировка методом «Вставка»;
* void QuickSort() – быстрая сортировка;
* int Find(T a) – метод поиска вхождения элемента в вектор, возвращает первый индекс вхождения;
* void FindAll(T a) – метод поиска всех вхождений элемента в вектор.

Единственный protected-метод:

* void qs(int first, int last) – внутренний метод для работы быстрой сортировки, содержит сам алгоритм.

## 3.3. Описание алгоритмов

Операция сложение матриц определена только для матриц одинаковых размеров. На выходе получается матрица такого же размера, все элементы которой являются суммой соответствующих элементов складываемых матриц.

Операция умножения определена для матриц, если у левой матрица количество столбцов совпадает с количеством строк правой матрицы. Результатом является матрица размером строк левой матрицы и столбцов правой, каждый элемент которой равен сумме произведений элементов i-ой строки матрицы А на соответствующие элементы j-ого столбца матрицы В.

Деление матрицы на матрицу представляет из себя умножение матрицы А на обратную к матрице B.

Сортировка пузырьком — это самый простой алгоритм сортировки. Он проходит по массиву несколько раз, на каждом этапе перемещая самое большое значение из неотсортированных в конец массива.

Сортировка вставками работает, проходя по массиву и перемещая нужное значение в начало массива. После того, как обработана очередная позиция, мы знаем, что все позиции до нее отсортированы, а после нее — нет. Важный момент: сортировка вставками обрабатывает элементы массива по порядку.

Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Поскольку класс Vector – это публичный наследник класса Matrix, то для него не имеет смысла описывать арифметическую операцию сложения, так как эта операция в числе других наследуется от класса Matrix.

# 4. Эксперименты

Оценим время, которое занимают матричные и векторно-матричные операции, с помощью асимптотической сложности. Рассмотрим код, отвечающий за суммирование матриц:

TMatrix<P> result(lhs.rows, lhs.columns);

for (int i = 0; i < lhs.rows; i++)

{

for (int j = 0; j < lhs.columns; j++)

{

result[i][j] = lhs[i][j] + rhs[i][j];

}

}

Асимптотическая сложность этого кода:

Теперь произведем замеры времени сложения квадратных матриц (чтобы точнее соотнести результаты замеров с асимптотической сложностью):

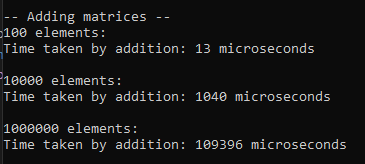


Рисунок 5: Замеры времени, полученные при выполнении сложения матриц (микросек.).

На основе полученных данных, построим таблицу соотношения размеров матриц и времени их суммирования:

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (микросек.) |
| 100 | 13 |
| 10000 | 1040 |
| 1000000 | 109396 |

Таблица 1: Результаты замеров времени суммирования матриц.

Заметим, что при увеличении количества элементов в 100 раз относительно предыдущего, время увеличивается примерно в 100 раз.

Теперь аналогично оценим асимптотическую сложность матричного умножения:

for (int i = 0; i < lhs.rows; i++)

for (int j = 0; j < rhs.columns; j++)

{

t = 0;

for (int k = 0; k < lhs.columns; k++)

t += lhs[i][k] \* rhs[k][j];

result[i][j] = t;

}

Асимптотическая сложность умножения:

Теперь произведем замеры времени умножения квадратных матриц:

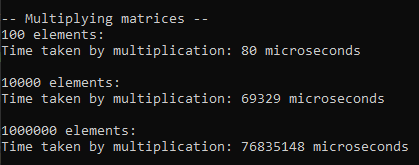


Рисунок 6: Замеры времени, полученные при выполнении умножения матриц (микросек.).

На основе полученных данных построим таблицу соотношения размеров матриц и времени умножения:

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (микросек.) |
| 100 | 80 |
| 10000 | 69329 |
| 1000000 | 76835148 |

Таблица 2: Результаты замеров времени умножения матриц.

Заметим, что при увеличении количества элементов в 100 раз относительно предыдущего, время увеличивается примерно в 1000 раз.

Для стандартных алгоритмов сортировки (BubbleSort, IsertionSort и QuickSort) асимптотическая сложность уже определена. Следовательно, достаточно произвести замеры и составить таблицу:

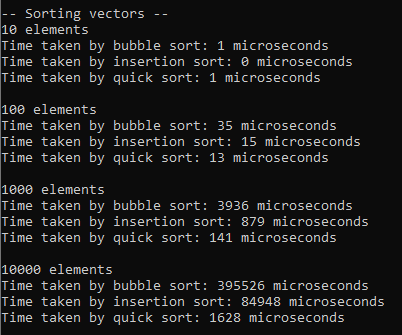


Рисунок 7: Замеры времени, полученные при выполнении сортировки вектора (в сек.)

Полученные данные занесем в таблицу:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вид сортировки | Асимптотическая сложность | Время (микросек.) на кол-во элементов | | | |
| 10 | 100 | 1000 | 10000 |
| BubbleSort |  | 1 | 35 | 3936 | 395526 |
| InsertionSort |  | 0 | 15 | 879 | 84948 |
| QuickSort |  | 1 | 13 | 141 | 1628 |

Таблица 4: Результаты сортировок по времени для массивов

Можно заметить, что затрачиваемое на реализацию операции время растет эквивалентно асимптотической функции.

# Заключение

Таким образом, в рамках данной лабораторной работы была успешно создана программа с использованием шаблонов, реализующая классы для работы с векторами и матрицами. Эксперименты показывают, что с некоторыми погрешностями, затраченное на реализацию матричных операций и сортировок время совпадает с ожидаемым относительно асимптотической сложности временем.

# Литература

1. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003.
2. Официальный сайт Habr. – Режим доступа <https://habr.com/ru/post/339656/>
3. Официальный сайт Microsoft. – Режим доступа <https://docs.microsoft.com/ru-ru/cpp/c-language/main-function-and-program-execution?view=vs-2019>
4. Сайт Stackoverflow.com. – Режим доступа <https://stackoverflow.com/>

# Приложение

***Код программы***

### main.cpp

#include "Matrix.h"

#include "Vector.h"

int main()

{

srand(time(NULL));

cout << "-- Creating a 3x3 matrix and testing = and the copy constructor --" << endl;

TMatrix<int> m(3,3);

TMatrix<int> l(m);

cout << l << endl;

m = l;

cout << m;

if (l == m)

cout << "Matrices are equal, successful" << endl;

else

cout << "Matrices aren't equal" << endl;

cout << endl << "Type in a 3x3 matrix: " << endl;

cin >> m;

m.set\_size(2, 2);

cout << endl << "Shrinked!" << endl << m << endl;

m = m \* 5;

cout << "Multiplied by " << 5 << endl << m << endl;

cout << "-- Testing vectors --" << endl;

TVector<int> ah(5);

ah[0] = 15;

cout << ah;

TVector<int> bleh{ 1, 2, 3, 4 };

ah = { 10, 20, 30, 40 };

bleh = bleh + ah;

cout << bleh;

if (bleh == l)

cout << "Vectors are equal" << endl;

else

cout << "Vectors aren't equal" << endl;

cout << endl << "-- Multiplying matrices --" << endl;

int n = 10;

for (int i = 1; i < 4; i++)

{

cout << n\*n << " elements: " << endl;

TMatrix<int> RandM(n, n, 1, 100);

//TMatrix<int> RandN(n, n, 1, 100);

//RandM + RandN;

TMatrix<int> RandO(n, n, 1, 100);

RandM\* RandO;

cout << endl;

n = n \* 10;

}

cout << "-- Sorting vectors --" << endl;

n = 10;

for (int i = 1; i < 5; i++)

{

TVector<double> ah(n, 1, 100);

TVector<double> bh(ah);

cout << ah.get\_columns() << " elements" << endl;

ah.BubbleSort();

ah = bh;

ah.InsertSort();

ah = bh;

ah.QuickSort();

ah.~TVector();

bh.~TVector();

cout << endl;

n = n \* 10;

}

cout << "Testing Find() and FindAll()" << endl;

TVector<int> Vec = { 1, 2, 3, 4, 5, 2, 2, 2, 2 };

cout << endl << "Item 0: [" << Vec.Find(0) << "], item 3: [" << Vec.Find(3) << "]" << endl << endl;

cout << "Item 2: " << endl;

Vec.FindAll(2);

cout << endl;

return 0;

}

### Matrix.h

#pragma once

#include <chrono>

#include <random>

#include <iostream>

using namespace std;

using namespace std::chrono;

template <class T>

class TMatrix

{

protected:

int rows, columns;

T\*\* matrix;

void inversion();

public:

TMatrix() :rows(0), columns(0), matrix(0) {};

TMatrix(int \_r, int \_c);

TMatrix(const TMatrix& rhs);

TMatrix(int \_r, int \_c, int a, int b);

virtual ~TMatrix();

int get\_rows() const;

int get\_columns() const;

T\*\* get\_array() const;

void set\_size(int new\_r, int new\_c);

const TMatrix<T> &operator =(const TMatrix<T>& rhs);

T\*& operator [](const int i) const;

template<class P>

friend TMatrix<P> operator +(TMatrix<P>& lhs, TMatrix<P>& rhs);

template <class M>

friend TMatrix<M> operator -(TMatrix<M>& lhs, TMatrix<M>& rhs);

template <class E>

friend bool operator ==(TMatrix<E>& lhs, TMatrix<E>& rhs);

template <class M>

friend TMatrix<M> operator \*(TMatrix<M>& lhs, TMatrix<M>& rhs);

template <class M, typename N>

friend TMatrix<M> operator \*(TMatrix<M>& lhs, N rhs);

template <class D, typename N>

friend TMatrix<D> operator /(TMatrix<D>& lhs, N rhs);

template <class I>

friend istream& operator >>(istream& in, TMatrix<I>& t);

template <class O>

friend ostream& operator <<(ostream& out, const TMatrix<O>& t);

};

#include "Matrix\_imp.h"

### Matrix\_imp.h

#include "Matrix.h"

template<class T>

inline TMatrix<T>::TMatrix(int \_r, int \_c)

{

matrix = new T \* [\_r];

for (int i = 0; i < \_r; i++)

{

matrix[i] = new T[\_c];

for (int j = 0; j < \_c; j++)

matrix[i][j] = 0;

}

rows = \_r;

columns = \_c;

}

template<class T>

inline TMatrix<T>::TMatrix(const TMatrix& rhs)

{

matrix = new T \* [rhs.rows];

for (int i = 0; i < rhs.rows; i++)

{

matrix[i] = new T[rhs.columns];

for (int j = 0; j < rhs.columns; j++)

matrix[i][j] = rhs[i][j];

}

rows = rhs.rows;

columns = rhs.columns;

}

template<class T>

inline TMatrix<T>::TMatrix(int \_r, int \_c, int a, int b)

{

matrix = new T \* [\_r];

for (int i = 0; i < \_r; i++)

{

matrix[i] = new T[\_c];

for (int j = 0; j < \_c; j++)

matrix[i][j] = ((double)rand() / (double)RAND\_MAX) \* (b - a) + a;

}

rows = \_r;

columns = \_c;

}

template<class T>

inline TMatrix<T>::~TMatrix()

{

if ((rows != 0) & (columns != 0))

{

for (int i = 0; i < rows; i++) delete[] matrix[i];

delete[] matrix;

rows = 0;

columns = 0;

}

}

template<class T>

inline int TMatrix<T>::get\_rows() const

{

return rows;

}

template<class T>

inline int TMatrix<T>::get\_columns() const

{

return columns;

}

template<class T>

inline T\*\* TMatrix<T>::get\_array() const

{

return matrix;

}

template<class T>

const TMatrix<T>& TMatrix<T>::operator=(const TMatrix<T>& rhs)

{

if (this == &rhs)

return \*this;

for (int i = 0; i < rows; i++) delete[] matrix[i];

delete[] matrix;

rows = rhs.rows;

columns = rhs.columns;

matrix = new T \* [rows];

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

{

matrix[i] = new T[columns];

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

{

matrix[i][j] = rhs[i][j];

}

}

return \*this;

}

template<class T>

inline T\*& TMatrix<T>::operator[](const int i) const

{

if (i > rows || i < 0)

throw length\_error("Incorrect index");

return matrix[i];

}

template<class P>

inline TMatrix<P> operator+(TMatrix<P>& lhs, TMatrix<P>& rhs)

{

if (lhs.rows != rhs.rows || lhs.columns != rhs.columns)

throw "Matrices aren't equal";

auto start = high\_resolution\_clock::now();

TMatrix<P> result(lhs.rows, lhs.columns);

for (int i = 0; i < lhs.rows; i++)

{

for (int j = 0; j < lhs.columns; j++)

{

result[i][j] = lhs[i][j] + rhs[i][j];

}

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken by addition: "

<< duration.count() << " microseconds" << endl;

return result;

}

template<class M>

inline TMatrix<M> operator-(TMatrix<M>& lhs, TMatrix<M>& rhs)

{

if (lhs.rows != rhs.rows || lhs.columns != rhs.columns)

throw "Matrices aren't equal";

TMatrix<M> result(lhs.rows, lhs.columns);

for (int i = 0; i < lhs.rows; i++)

{

for (int j = 0; j < lhs.columns; j++)

{

result[i][j] = lhs[i][j] - rhs[i][j];

}

}

return result;

}

template<class E>

inline bool operator==(TMatrix<E>& lhs, TMatrix<E>& rhs)

{

if (lhs.rows != rhs.rows || lhs.columns != rhs.columns)

{

return false;

}

else {

for (int i = 0; i < lhs.rows; i++)

{

for (int j = 0; j < lhs.columns; j++)

{

if (lhs[i][j] != rhs[i][j]) return false;

}

}

}

return true;

}

template<class M>

inline TMatrix<M> operator\*(TMatrix<M>& lhs, TMatrix<M>& rhs)

{

if (lhs.columns != rhs.rows)

throw "Can't multiply";

auto start = high\_resolution\_clock::now();

TMatrix<M> result(lhs.rows, rhs.columns);

M t;

for (int i = 0; i < lhs.rows; i++)

for (int j = 0; j < rhs.columns; j++)

{

t = 0;

for (int k = 0; k < lhs.columns; k++)

t += lhs[i][k] \* rhs[k][j];

result[i][j] = t;

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken by multiplication: "

<< duration.count() << " microseconds" << endl;

return result;

}

template<class M, typename N>

inline TMatrix<M> operator\*(TMatrix<M>& lhs, N rhs)

{

TMatrix<M> result(lhs.rows, lhs.columns);

for (int i = 0; i < lhs.rows; i++)

for (int j = 0; j < lhs.columns; j++)

result[i][j] = lhs[i][j] \* rhs;

return result;

}

template<class D, typename N>

inline TMatrix<D> operator/(TMatrix<D>& lhs, N rhs)

{

TMatrix<D> result(lhs.rows, lhs.columns);

for (int i = 0; i < lhs.rows; i++)

for (int j = 0; j < lhs.columns; j++)

result[i][j] = lhs[i][j] / rhs;

return result;

}

template<class I>

inline istream& operator>>(istream& in, TMatrix<I>& t)

{

for (int i = 0; i < t.rows; i++)

{

for (int j = 0; j < t.columns; j++)

{

in >> t[i][j];

}

}

return in;

}

template<class O>

inline ostream& operator<<(ostream& out, const TMatrix<O>& t)

{

for (int i = 0; i < t.rows; i++)

{

for (int j = 0; j < t.columns; j++)

{

out << t[i][j] << " ";

}

out << endl;

}

return out;

}

template<class T>

inline void TMatrix<T>::set\_size(int new\_r, int new\_c)

{

if ((new\_r < 0 || new\_c < 0) & (new\_r != -1) & (new\_c != -1))

throw length\_error("Incorrect size");

TMatrix<T> D(\*this);

for (int i = 0; i < rows; i++) delete[] matrix[i];

delete[] matrix;

if (new\_r != -1)

rows = new\_r;

if (new\_c != -1)

columns = new\_c;

matrix = new T \* [rows];

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

matrix[i] = new T[columns];

for (int i = 0; i < rows; i++) {

for (int j = 0; j < columns; j++) {

matrix[i][j] = 0;

}

}

int wb = (D.rows < rows) ? D.rows : rows;

int lb = (D.columns < columns) ? D.columns : columns;

for (int i = 0; i < wb; i++) {

for (int j = 0; j < lb; j++) {

matrix[i][j] = D[i][j];

}

}

}

template<class T>

void TMatrix<T>::inversion()

{

double temp;

int N = rows;

T\*\* E = new T\* [N];

for (int i = 0; i < N; i++)

E[i] = new T[N];

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

{

E[i][j] = 0.0;

if (i == j)

E[i][j] = 1.0;

}

for (int k = 0; k < N; k++)

{

temp = matrix[k][k];

for (int j = 0; j < N; j++)

{

matrix[k][j] /= temp;

E[k][j] /= temp;

}

for (int i = k + 1; i < N; i++)

{

temp = matrix[i][k];

for (int j = 0; j < N; j++)

{

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

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

}

}

}

for (int k = N - 1; k > 0; k--)

{

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

{

temp = matrix[i][k];

for (int j = 0; j < N; j++)

{

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

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

}

}

}

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

matrix[i][j] = E[i][j];

for (int i = 0; i < N; i++)

delete[] E[i];

delete[] E;

}

template<class D>

inline TMatrix<D> operator/(TMatrix<D>& lhs, TMatrix<D>& rhs)

{

if ((lhs.get\_rows() == rhs.get\_rows()) & (lhs.get\_columns() == rhs.get\_rows()))

{

TMatrix<D> I = rhs;

I.inversion();

return lhs \* I;

}

else

throw "Yikes";

}

### Vector.h

#pragma once

#include "Matrix.h"

using namespace std::chrono;

template <class T>

class TVector :public TMatrix<T>

{

public:

TVector() :TMatrix<T>() {}

TVector(int \_size) :TMatrix<T>(1, \_size) {}

TVector(const TVector& t) : TMatrix<T>(t) {}

virtual ~TVector();

TVector(const std::initializer\_list<T>& list);

TVector(int n, int a, int b);

const TVector<T>& operator=(const std::initializer\_list<T>& list);

const TVector<T>& operator =(const TMatrix<T>& rhs);

template<class P, typename N>

friend TVector<N> operator +(TVector<P>& lhs, TVector<N>& rhs);

template <class M, typename N>

friend TVector<N> operator -(TVector<M>& lhs, TVector<N>& rhs);

template <class D, typename N>

friend TVector<D> operator /(TVector<D>& lhs, N rhs);

template <class M, typename N>

friend TVector<M> operator \*(TMatrix<M>& lhs, TVector<N>& rhs);

template <class M, typename N>

friend TMatrix<M> operator \*(TVector<M>& lhs, TMatrix<N>& rhs);

T\* get\_vector();

T& operator [](const int i) const;

template <class I>

friend istream& operator >>(istream& in, TVector<I>& t);

template <class O>

friend ostream& operator <<(ostream& out, const TVector<O>& t);

void BubbleSort();

void InsertSort();

void QuickSort();

int Find(T a);

void FindAll(T a);

protected:

void qs(int first, int last);

};

#include "Vector\_imp.h"

### Vector\_imp.h

#include "Vector.h"

template<class T>

inline TVector<T>::~TVector()

{

if ((this->rows != 0) & (this->columns != 0))

{

delete[] this->matrix[0];

delete[] this->matrix;

this->rows = 0;

this->columns = 0;

//cout << "deleted";

}

}

template<class T>

TVector<T>::TVector(const std::initializer\_list<T>& list)

: TVector(list.size())

{

int count = 0;

for (auto& element : list)

{

this->matrix[0][count] = element;

++count;

}

}

template<class T>

inline TVector<T>::TVector(int n, int a, int b)

{

this->matrix = new T \* [1];

this->matrix[0] = new T[n];

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

this->matrix[0][i] = ((double)rand() / (double)RAND\_MAX) \* (b - a) + a;

this->rows = 1;

this->columns = n;

}

template<class T>

inline T\* TVector<T>::get\_vector()

{

return this->matrix;

}

template<class T>

inline T& TVector<T>::operator[](const int i) const

{

return this->matrix[0][i];

}

template<class T>

const TVector<T>& TVector<T>::operator=(const std::initializer\_list<T>& list)

{

delete[] this->matrix[0];

delete[] this->matrix;

this->matrix = new T \* [1];

this->matrix[0] = new T[list.size()];

this->columns = list.size();

int count = 0;

for (auto& element : list)

{

this->matrix[0][count] = element;

++count;

}

return \*this;

}

template<class T>

inline const TVector<T>& TVector<T>::operator=(const TMatrix<T>& rhs)

{

if (this == &rhs)

return \*this;

delete[] this->matrix[0];

delete[] this->matrix;

this->rows = 1;

this->columns = rhs.get\_columns();

this->matrix = new T \* [1];

this->matrix[0] = new T[this->columns];

for (int i = 0; i < this->columns; i++)

{

this->matrix[0][i] = rhs[0][i];

}

return \*this;

}

template<class P, typename N>

inline TVector<N> operator+(TVector<P>& lhs, TVector<N>& rhs)

{

if (lhs.rows != rhs.rows || lhs.columns != rhs.columns)

throw "Vectors aren't equal";

TVector<N> result(lhs.columns);

for (int i = 0; i < lhs.columns; i++)

{

result[i] = lhs[i] + rhs[i];

}

return result;

}

template<class M, typename N>

inline TVector<N> operator-(TVector<M>& lhs, TVector<N>& rhs)

{

if (lhs.rows != rhs.rows || lhs.columns != rhs.columns)

throw "Vectors aren't equal";

TVector<N> result(lhs.columns);

for (int i = 0; i < lhs.columns; i++)

{

result[i] = lhs[i] - rhs[i];

}

return result;

}

template<class M, typename N>

inline TVector<M> operator\*(TMatrix<M>& lhs, TVector<N>& rhs)

{

if (lhs.get\_columns() != rhs.columns)

throw "Can't multiply";

TVector<M> result(lhs.get\_rows());

M t;

for (int i = 0; i < rhs.columns; i++)

{

t = 0;

for (int j = 0; j < lhs.get\_columns(); j++)

t += lhs[i][j] \* rhs[j];

result[i] = t;

}

return result;

}

template<class M, typename N>

inline TMatrix<M> operator\*(TVector<M>& lhs, TMatrix<N>& rhs)

{

if (lhs.rows != rhs.get\_rows())

throw "Can't multiply";

TMatrix<M> result(lhs.columns, rhs.get\_columns());

for (int i = 0; i < lhs.columns; i++)

for (int j = 0; j < rhs.get\_columns(); j++)

{

result[i][j] = lhs[i] \* rhs[0][j];

}

return result;

}

template<class D, typename N>

inline TVector<D> operator/(TVector<D>& lhs, N rhs)

{

TVector<D> result(lhs.columns);

for (int i = 0; i < lhs.columns; i++)

result[i] = lhs[i] / rhs;

return result;

}

template<class I>

inline istream& operator>>(istream& in, TVector<I>& t)

{

for (int i = 0; i < t.columns; i++)

in >> t[i];

return in;

}

template<class O>

inline ostream& operator<<(ostream& out, const TVector<O>& t)

{

for (int i = 0; i < t.columns; i++)

{

out << t[i] << endl;

}

out << endl;

return out;

}

template<class T>

inline void TVector<T>::BubbleSort()

{

auto start = high\_resolution\_clock::now();

for (int i = 0; i < this->columns; i++)

{

for (int j = i + 1; j < this->columns; j++)

{

if (this->matrix[0][i] > this->matrix[0][j])

{

T element;

element = this->matrix[0][i];

this->matrix[0][i] = this->matrix[0][j];

this->matrix[0][j] = element;

}

}

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken by bubble sort: "

<< duration.count() << " microseconds" << endl;

}

template<class T>

inline void TVector<T>::InsertSort()

{

int begin;

int end;

int middle;

T value;

auto start = high\_resolution\_clock::now();

for (int i = 1; i < this->columns; ++i)

{

value = this->matrix[0][i];

begin = 0;

end = i - 1;

while (begin <= end)

{

middle = (begin + end) / 2;

if (value < this->matrix[0][middle])

{

end = middle - 1;

}

else

{

begin = middle + 1;

}

}

for (int j = i - 1; j >= begin; --j)

{

this->matrix[0][j + 1] = this->matrix[0][j];

}

this->matrix[0][begin] = value;

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken by insertion sort: "

<< duration.count() << " microseconds" << endl;

}

template<class T>

inline void TVector<T>::qs(int first, int last)

{

int i = first, j = last;

T tmp, x = this->matrix[0][(first + last) / 2];

do {

while (this->matrix[0][i] < x)

i++;

while (this->matrix[0][j] > x)

j--;

if (i <= j)

{

if (i < j)

{

tmp = this->matrix[0][i];

this->matrix[0][i] = this->matrix[0][j];

this->matrix[0][j] = tmp;

}

i++;

j--;

}

} while (i <= j);

if (i < last)

qs(i, last);

if (first < j)

qs(first, j);

}

template<class T>

inline void TVector<T>::QuickSort()

{

auto start = high\_resolution\_clock::now();

this->qs(0, this->columns - 1);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken by quick sort: "

<< duration.count() << " microseconds" << endl;

}

template<class T>

inline int TVector<T>::Find(T a)

{

for (int i = 0; i < this->columns; i++)

if (this->matrix[0][i] == a)

return i;

return -1;

}

template<class T>

inline void TVector<T>::FindAll(T a)

{

int k = 0;

for (int i = 0; i < this->columns; i++)

if (this->matrix[0][i] == a)

{

cout << "Item found at: [" << i << "]" << endl;

k = 1;

}

if (k == 0)

cout << "item not found" << endl;

}