Федеральное агентство по образованию Российской Федерации

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

Институт информационных технологий, математики и механики

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчёт по лабораторной работе

**Работа с классом «Вектор»**

Выполнила:

студентка ф-та ИИТММ

Гордей М.В

Проверил:

ассистент к.математического обеспечения и суперкомпьютерных технологий

Лебедев И.Г.

Нижний Новгород

2020 г.

Содержание

1.[Введение 3](#_Toc24469347)

2.[Постановка задачи 4](#_Toc24469348)

3[Руководство пользователя 5](#_Toc24469349)

4[Руководство программиста 6](#_Toc24469350)

4.1.[Описание структуры программы 6](#_Toc24469351)

4.2[Описание структур данных 6](#_Toc24469352)

4.3[Описание алгоритмов 6](#_Toc24469353)

5.[Эксперименты 7](#_Toc24469354)

6.[Заключение 8](#_Toc24469355)

7.[Литература 9](#_Toc24469356)

8.[Приложения 10](#_Toc24469357)

[Приложение 1 10](#_Toc24469358)

# 1.Введение

Матрицы и векторы – важнейшие понятия в курсе линейной алгебры. Матрица в математическом понимании – объект, записываемый в виде прямоугольной таблицы, которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Векторами называются матрицы, состоящие из одной строки или одного столбца.

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.

Для организации работы с матрицами и векторами были написаны классы, осуществляющие выполнение математических операций. Данная работа демонстрирует реализацию классов для работы с матрицами и векторами с использованием шаблонов.

# 2.Постановка задачи

Написать классы для работы с векторами и матрицами использовать шаблоны.

Продемонстрировать их работу на примере (написать в main пример).

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);

Также в классе вектор должна быть возможность отсортировать его тремя способами (пузырек, вставка, быстрая сортировки). Также нужно сравнить время работы, сделать выводы.

# 3.Руководство пользователя

Использование программы пользователем, при открытии приложения

1. Создать математические объекты (вектор или матрицу)
2. Расширить объекты при помощи доступа к защищенным полям(гетеры) или при помощи конструкторов
3. Выполнить необходимые математические операции, производимые с векторами и матрицами, применимые в алгебре (умножение, сложение, деление, вычитание матриц и векторов при подходящем размере, умножение вектора на скаляр, умножение вектора на матрицу и наоборот при помощи матричного умножения), проверка работы перегруженных операций, конструкторов.

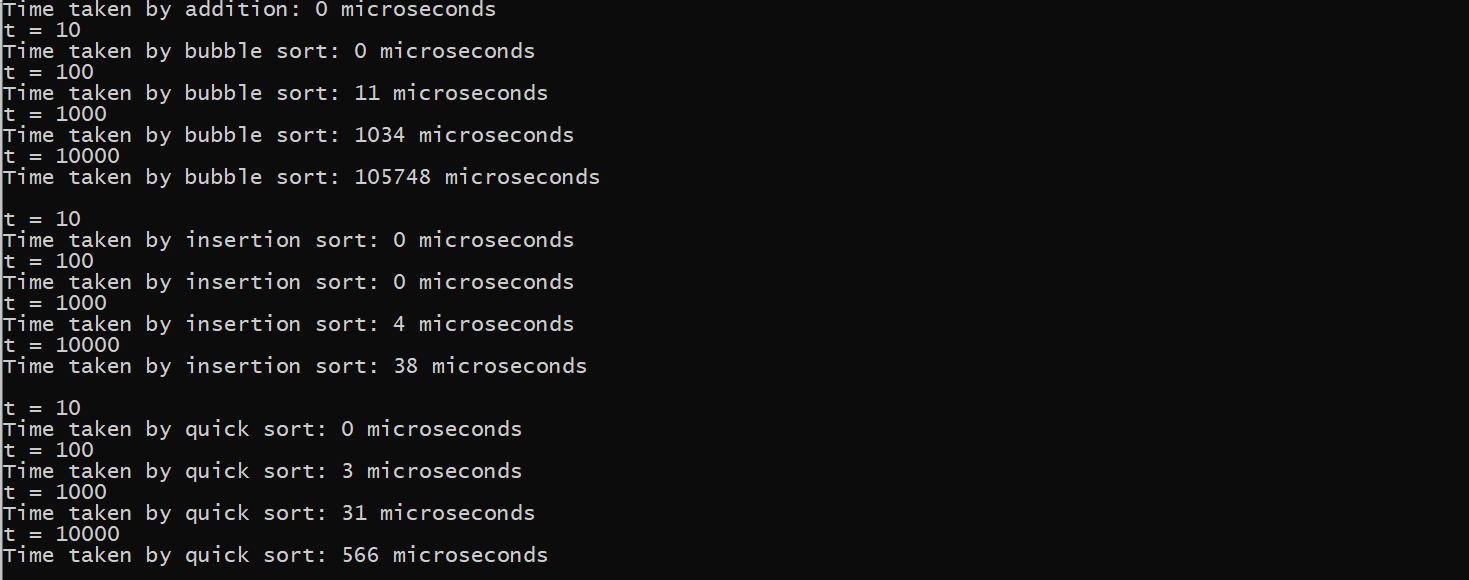


Рисунок 1: интерфейс работы программы(данные, необходимые для экспериментов в работе)

**4.Руководство программиста**

**4.1. Описание структуры программы**

Программа состоит из одного решения.

В решении Vector определено 4 модуля main.cpp, Matrix.h, Vector\_i.h, Matrix\_i.h, Vector.h

* В модуле main.cpp определена стандартная функция int main(), внутри которой содержится набор действий с объектами типа Matrix и Vector, где продемонстрирована работа программы.
* В модуле Matrix.h определен класс Matrix,
* В модуле Matrix\_i.h объявлены все методы класса матриц и их определения, объявлены и реализованы функции перегрузки операторов матрично-векторного умножения.
* В модуле Vector.h определен класс Vector,
* В модуле Vector.h также объявлены все методы и их определения класса векторов.

**4.2. Описание структур данных**

В программе определены два следующих шаблонных класса:

Class Matrix

Class Vector

Внутри класса Vector определены следующие поля:

T\* arr шаблонный указатель;

size\_t n длина вектора;

Внутри класса Vector определен следующий набор public-методов

(плюс конструкторы и деструктор):

* Vector() – конструктор по умолчанию, не принимает никаких параметров, инициализирует все поля 0, через списки инициализации;
* Vector(int size, const T\* \_arr) – конструктор инциализатор, принимает на вход параметр типа int и шаблонный указатель, создает динамический массив заданного размера, заполняет его теми же элементами какими заполнен переданный указатель;
* Vector(const\_t size) - конструктор инциализатор, принимает на вход параметр типа t\_size, создает динамический массив заданного размера
* Vector(const Vector<T>& other) – конструктор копирования, принимает на вход объект типа Vector, создает объект с теми же характеристиками что и переданный;
* ~Vector() – деструктор, очищает выделенную методами и конструкторами память;
* int GetN() – возвращает длину вектора;
* T operator [](size\_t index) – перегрузка оператора индексации, позволяет возвращать элемент вектора по индексу;
* Vector<T> operator+(const Vector<T>& other) – перегрузка оператора суммы, позволяет возвращать вектор, являющийся суммой двух векторов;
* Vector<T> operator-(const Vector<T>& other) – перегрузка оператора разности, позволяет возвращать вектор, являющийся разностью двух векторов;
* Vector<T> operator = (const Vector<T>& other) – перегрузка оператора присвоить, позволяет присваивать один объект типа Vector другому;
* bool operator ==(const Vector<T>& other) – перегрузка оператора сравнения, позволяет сравнивать объекты типа Vector;
* T operator\*(const Vector<T>& other) – перегрузка оператора умножить, позволяет перемножать объекты типа Vector;
* friend ostream& operator<<(std::ostream& stream, const Vector<Z>& self)– перегрузка оператора вывода, позволяет выводить вектора на экран;
* friend istream& operator>>(std::istream& stream, Vector<Z>& self) – перегрузка оператора ввода, позволяет вводить вектора;
* void bubblesort() - сортировка методом «Пузырек».Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее;
* Void insertionsort () - сортировка методом «Вставка».Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее;
* void quicksort(int first, int last) – быстрая сортировка.Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее;

**4.2. Описание структур данных**

В программе определены два следующих шаблонных класса:

Class Matrix

Class Vector

Внутри класса Matrix определены следующие поля:

* T\*\* M – шаблонный двойной указатель;
* size\_t n – количество строк матрицы;
* size\_t m– количество столбцов матрицы;

Внутри класса Matrix определен следующий набор public-методов

(плюс конструкторы и деструктор):

* Matrix() – конструктор по умолчанию, не принимает никаких параметров, инициализирует все три поля 0, через списки инициализации;
* Matrix( size\_t size1, size\_t size2, T\*\* M\_) – конструктор инциализатор, принимает на вход два параметра типа size\_t и один шаблонный двойной указатель, создает динамический массив заданных размеров, заполняет его теми же элементами какими заполнен переданный указатель
* Matrix(size\_t size1, size\_t size2) - конструктор инциализатор, принимает на вход два параметра типа size\_t , создает динамический массив заданного размера, заполняет его случайными числами.
* Matrix(const Matrix<T> & other) – конструктор копирования, принимает на вход объект типа Matrix, создает объект с теми же характеристиками что и переданный, и матрицей того же содержимого;
* ~Matrix() – деструктор, очищает выделенную методами и конструкторами память;
* int GetN() – возвращает количество строк матрицы;
* int GetM() – возвращает количество столбцов матрицы;
* T\*& operator [](const int \_n) – перегрузка оператора индексации, позволяет возвращать элемент матрицы по индексу;
* Matrix<T> operator+(const Matrix<T>& other) – перегрузка оператора суммы, позволяет возвращать матрицу, являющуюся суммой двух матриц;
* Matrix<T> operator-(const Matrix<T>& other) – перегрузка оператора разности, позволяет возвращать матрицу, являющуюся разностью двух матриц;
* Matrix<T> operator = (const Matrix<T>& other) – перегрузка оператора присвоить, позволяет присваивать один объект типа Matrix другому;
* bool operator ==(const Matrix<T>& other) – перегрузка оператора сравнения, позволяет сравнивать объекты типа Matrix;
* Matrix<T> operator\*(const Matrix<T>& other) – перегрузка оператора умножить, позволяет перемножать обьекты типа Matrix;
* friend ostream& operator<<( (std::istream& stream, Matrix<T>& self) – перегрузка оператора вывода, позволяет выводить матрицы на экран;
* friend istream& operator>>(std::ostream& stream, const Matrix<T>& self) – перегрузка оператора ввода, позволяет вводить матрицы.

**4.3. Описание алгоритмов**

Опишем алгоритмы сортировки в данной программе:

В данной лабораторной работе реализованы соответствующие алгоритмы сортировок:

1)Алгоритм сортировки пузырьком

Шаг 1. Пусть J:=1 , т.е. итоговый участок состоит из одного элемента.

Шаг 2. Берется первый элемент остатка и перемещается на место в итоговый участок массива так, чтобы итог остался упорядоченным. Первый элемент остатка назовем перемещаемым. Перемещение выполняется путем сравнения перемещаемого элемента с предшествующим ему элементом. Если предшествующий элемент меньше сравниваемого элемента, то процесс перемещения закончен. В противном случае сравниваемые элементы переставляются и, если элемент не достиг начала массива, то повторяется Шаг

2. Шаг 3. После того, как первый элемент остатка переместился в итоговый участок, увеличивается на единицу значение переменной J, тем самым увеличивая отсортированную часть массива. Если J Если J<N, то управление передается на Шаг 2, в противном случае - работа алгоритма завершена.

Конец алгоритма

2)Сортировка вставками

Алгоритм метода включения

Шаг 1. Пусть J=1 , т.е. итоговый участок состоит из одного элемента.

Шаг 2. Берется первый элемент остатка и перемещается в отсортированную часть массива

так, чтобы итоговый участок остался упорядоченным. Делается это с помощью

обращения к процедуре двоичного поиска, которая в качестве выходного

параметра дает номер элемента массива, на месте которого должен бы находиться

перемещаемый элемент. Если этот номер указывает на место в итоговом участке

массива, то сдвигаются все элементы итогового участка массива, начиная с этого

номера на одно место вправо, а перемещаемый элемент ставится на

освободившееся место.

Шаг 3. После того, как первый элемент остатка переместился в итоговый участок,

увеличивается на единицу значение переменной J, тем самым увеличивая

отсортированную часть массива. Если J < N, то управление передается на Шаг 2, в

противном случае - работа алгоритма завершена.

Конец алгоритма.

3)Алгоритм быстрой сортировки

Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Алгоритм сложения/вычитания матриц:

**Суммой (разностью) матриц** A и B одного размера называется матрица C=A+BC=A+B такого же размера, получаемая из исходных путем сложения соответствующих элементов. 

На входе проверка на совпадение размеров матрицы, создание новой матрицы для записи результата, если операция определена. В цикле элементы матрицы поэлементно складываются или вычитаются, в конце возвращается матица результата.

Алгоритм суммирования или вычитания векторов

Сначала выполняется проверка на совпадение размерности. Для сложения векторов необходимо сложить соответствующие координаты этих векторов. Создается вектор для записи результата. По циклу соответствующие координаты складываются или вычитаются, в результате работы возвращается результирующий вектор.

Алгоритм умножения матриц:

Если чисто столбцов в первой матрице совпадает с числом строк во второй, то эти матрицы можно перемножить. На входе выполняется эта проверка. Создается матрица для записи результата. В цикле каждая строка первой матрицы почленно умножается с каждым столбцом второй матрицы. Возвращается результирующая матрица.

Алгоритмы нахождения произведения вектора на матрицу и матрицы на вектор аналогичны алгоритму умножения матриц.

**5.Эксперименты**

Оценим время, которое занимают матричные, векторно-матричные операции, с помощью асимптотической сложности.

Matrix<T> Matrix<T>::operator +(const Matrix& other)

{

if ((this->m != other.m) || (this->n != other.n))

throw length\_error("-1");

for (int k = 0; k < n; k++) {

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

(\*this).M[k][i] = (\*this).M[k][i] + other.M[k][i];

}

}

return \*this;

}

Асимптотическая сложность этого кода:

Теперь мы произведем замеры времени умножения матриц

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения ( микросек.) |
| 10 | 0 |
| 100 | 39 |
| 1000 | 3616 |

Таблица 2: Время умножения матриц.

Как мы можем видеть действительно, при увеличении количества элементов в 10 раза относительно предыдущего, время увеличивается примерно 10 раз с учетом погрешностей (от сгенерированных чисел в матрице время работы когда может отличаться)

1. Теперь аналогично оценим асимптотическую сложность векторно-матричного умножения:

template <typename T>

Matrix<T> Matrix<T>::operator\*(const Matrix<T>& other)

{

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

if (this->m == other.n)

{

Matrix<T> result;

result.n = this->n;

result.m = other.m;

result.M = new T \* [result.n];

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

result.M[i] = new T[result.m];

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

for (int j = 0; j < result.m; j++)

{

result.M[i][j] = 0;

for (int k = 0; k < this->m; k++)

{

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

}

}

Асимптотическая сложность этого кода :

Теперь мы произведем замеры времени умножения матриц и векторов

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (микросек.) |
| 10 | 1 |
| 100 | 6 |
| 1000 | 4564 |

Таблица 3: Время умножения матриц и векторов

Как мы можем видеть действительно, при увеличении количества элементов в 2 раза относительно предыдущего, время увеличивается примерно 4 раза. Теперь перейдем к сортировкам. Асимптотическая сложность уже определена. Поэтому просто произведём замеры и составим таблицу.

С сортировками все значительнее проще, поскольку, для стандартных алгоритмов сортировки (BubbleSort, IsertionSort и QuickSort) Асимптотическая сложность уже определена. Поэтому просто произведём замеры и составим таблицу:

Внесем полученные данные в таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Вид сортировки | Асимптотическая сложность | Количество элементов | Время (мксек.) |
| BubbleSort |  | 10 | 0 |
| 100 | 11 |
| 10000 | 1035 |
| InsertionSort |  | 10 | 0 |
| 100 | 0 |
| 1000 | 4 |
| QuickSort |  | 10 | 0 |
| 100 | 3 |
| 1000 | 33 |

Таблица 4: Результаты сортировок по времени для массивов

Как видно из таблицы, действительно, затрачиваемое на реализацию операции время растет эквивалентно асимптотической функции.

**6.Заключение**

По итогу выполнения данной работы были написаны классы, позволяющие осуществлять удобную работу с матрицами и векторами. Проведенные в работе эксперименты показывают, что с некоторыми погрешностями, затраченное на реализацию матричных, матрично-векторных операций и сортировок время совпадает с ожидаемым относительно асимптотической сложности временем.

# 7.Литература

1.Официальный сайт Habr. – Режим доступа <https://habr.com/ru/post/339656/>

2.Керниган, Б.У. Язык программирования С / Б.У. Керниган, Д.М. Ритчи; Пер. с англ. В.Л. Бродовой. — М.: Вильямс, 2013. — 304 c.

3. Ашарина, И.В. Основы программирования на языках С и С++: Курс лекций для высших учебных заведений / И.В. Ашарина. — М.: Гор. линия-Телеком, 2012. — 208 c.

4. Иванова Г.С. Основы программирования: Учебник для вузов. – 2-е изд., перераб. и доп. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. – 416 с.

**8. Приложение**

Matrix.h

#include <exception>

#include <iostream>

#include "Vector.h"

using namespace std;

using namespace std::chrono;

template<typename T>

class Matrix

{

public:

Matrix() :M(0), n(0), m(0) {}

Matrix(const Vector<T>& v1);

Matrix(size\_t size1, size\_t size2, T\*\* M\_);

~Matrix();

Matrix operator +(const Matrix& other);

Matrix operator \*(const Matrix& other);

Matrix(int size1, int size2);

Matrix(const Matrix& other);

Matrix operator -(const Matrix& other);

Matrix& operator=(const Matrix& other);

bool operator==(const Matrix<T>& other);

const T\* operator [](size\_t i) const;

T\* operator [](size\_t i);

int GetN() const;

int GetM() const;

private:

T\*\* M;

size\_t n;

size\_t m;

};

#include "Matrix\_i.h"

Matrix\_i.h

template <typename T>

Matrix<T>::Matrix(const Vector<T>& v1)

{

n = v1.GetN();

m = 1;

M = new T \* [n];

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

M[i] = new T[m];

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

M[i][0] = v1[i];

}

template <typename T>

Matrix<T>::Matrix(size\_t size1, size\_t size2, T\*\* M\_)

{

n = size1;

m = size2;

M = new T \* [n];

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

M[i] = new T[m];

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

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

M[i][j] = M\_[i][j];

}

template <typename T>

Matrix<T>::~Matrix()

{

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

delete[] M[i];

delete[] M;

n = 0;

m = 0;

}

template <typename T>

Matrix<T> Matrix<T>::operator +(const Matrix& other)

{

if ((this->m != other.m) || (this->n != other.n))

throw length\_error("-1");

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

for (int k = 0; k < n; k++) {

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

(\*this).M[k][i] = (\*this).M[k][i] + other.M[k][i];

}

}

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

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

cout << "Time taken by addition: "

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

return \*this;

}

template <typename T>

Matrix<T>::Matrix(int size1, int size2)

{

n = size1;

m = size2;

M = new T \* [size1];

for (int i = 0; i < size1; i++)

{

M[i] = new T[size2];

for (int j = 0; j < size2; j++)

M[i][j] = 0;

}

}

template <typename T>

Matrix<T>::Matrix(const Matrix& other)

{

n = other.n;

m = other.m;

M = new T \* [n];

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

M[i] = new T[m];

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

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

M[i][j] = other.M[i][j];

}

template <typename T>

Matrix<T> Matrix<T>::operator -(const Matrix& other)

{

Matrix newmatrix;

for (int k = 0; k < n; k++) {

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

newmatrix[k][i] = M[k][i] - other.M[k][i];

}

}

return newmatrix;

}

template <typename T>

Matrix<T> operator \* (const Matrix<T>& x, const Vector<T>& y)

{

return x \* Matrix<T>(y);

}

template <typename T>

Matrix<T> operator \* (const Vector<T>& x, const Matrix<T>& y)

{

return Matrix<T>(x) \* y;

}

template <typename T>

int Matrix<T>::GetM() const

{

return m;

}

template <typename T>

int Matrix<T>::GetN() const

{

return n;

}

template <typename T>

Matrix<T>& Matrix<T>::operator=(const Matrix& other)

{

if (this == &other)

return \*this;

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

delete[] M[i];

delete[] M;

this->n = other.n;

this->m = other.m;

M = new T \* [n];

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

M[i] = new T[m];

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

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

M[i][j] = other.M[i][j];

return \*this;

}

template <typename T>

bool Matrix<T>::operator==(const Matrix<T>& other)

{

if (n == other.n && m == other.m)

{

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

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

if (M[i][j] != other.M[i][j])

return false;

return true;

}

else

return false;

}

template <typename T>

const T\* Matrix<T>::operator [](size\_t i) const

{

if (i >= n)

throw std::runtime\_error("wrong index");

else return M[i];

}

template <typename T>

T\* Matrix<T>::operator [](size\_t i)

{

if (i >= n)

throw std::runtime\_error("wrong index");

else return M[i];

}

template <typename T>

std::ostream& operator<<(std::ostream& stream, const Matrix<T>& self)

{

for (size\_t index = 0; index < self.GetN(); index++)

for (size\_t k = 0; k < self.GetM(); k++)

stream << self[index][k] << std::endl;

return stream;

}

template <typename T>

std::istream& operator>>(std::istream& stream, Matrix<T>& self)

{

size\_t size1 = 0;

size\_t size2 = 0;

stream >> size1;

stream >> size2;

Matrix <T> v4(size1, size2);

for (size\_t index = 0; index < size1; index++)

for (size\_t k = 0; k < size2; k++)

stream >> v4[index][k];

self = v4;

return stream;

}

template <typename T>

Matrix<T> Matrix<T>::operator\*(const Matrix<T>& other)

{

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

if (this->m == other.n)

{

Matrix<T> result;

result.n = this->n;

result.m = other.m;

result.M = new T \* [result.n];

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

result.M[i] = new T[result.m];

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

for (int j = 0; j < result.m; j++)

{

result.M[i][j] = 0;

for (int k = 0; k < this->m; k++)

{

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

}

}

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

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

cout << "Time taken by multiplication: "

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

return result;

}

else

throw length\_error("-1");

}

Vector.h

#pragma once

#include <exception>

#include <iostream>

#include <time.h>

#include <chrono>

#include <random>

using namespace std;

using namespace std::chrono;

template<typename T>

class Vector

{

public:

Vector();

Vector(const size\_t size);

T& operator [](size\_t index) const;

Vector(const Vector& other);

~Vector();

Vector(int num, int a, int b);

int GetN() const;

bool operator==(const Vector& other);

Vector& operator=(const Vector& other);

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

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

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

template <typename Z> friend std::ostream& operator<<(std::ostream& stream, const Vector<Z>& self);

template <typename Z> friend std::istream& operator>>(std::istream& stream, Vector<Z>& self);

void insertionsort();

void quicksort();

void bubblesort();

int Find(T a);

void FindAll(T a);

private:

T\* arr;

size\_t n;

void qs(int first, int last);

};

#include "Vector\_i.h"

Vector\_i/h

#include "Vector.h"

template <typename T>

Vector<T>::Vector() {};

template <typename T>

Vector<T>::Vector(const size\_t size)

{

n = size;

arr = new T[n]{};

}

template <typename T>

T& Vector<T>::operator [](size\_t index) const

{

if (index >= n || index < 0)

throw std::runtime\_error("wrong index");

return arr[index];

}

template<class T>

Vector<T>::Vector(int num, int a, int b)

{

arr = new T[num];

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

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

n = num;

}

template <typename T>

Vector<T>::Vector(const Vector& other)

{

n = other.n;

arr = new T[other.n]{};

for (size\_t index = 0; index < n; index++)

{

arr[index] = other.arr[index];

}

}

template <typename T>

Vector<T>::~Vector()

{

delete[] arr;

//arr = nullptr;

}

template <typename T>

bool Vector<T>::operator==(const Vector& other)

{

if (n != other.n)

return false;

for (size\_t index = 0; index < n; ++index)

{

if (arr[index] != other.arr[index])

return false;

}

return true;

}

template <typename T>

int Vector<T>::GetN() const

{

return n;

}

template <typename T>

Vector<T>& Vector<T>::operator=(const Vector& other)

{

delete[] arr;

arr = new T[other.GetN()];

n = other.GetN();

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

{

arr[i] = other[i];

}

return \*this;

}

template <typename T>

std::ostream& operator<<(std::ostream& stream, const Vector<T>& self)

{

for (size\_t index = 0; index < self.n; index++)

stream << self.arr[index] << std::endl;

return stream;

}

template <typename T>

std::istream& operator>>(std::istream& stream, Vector<T>& self)

{

size\_t size = 0;

stream >> size;

Vector <T> v4(size);

for (size\_t index = 0; index < size; index++)

stream >> v4[index];

self = v4;

return stream;

}

template <typename T>

void Vector<T>::insertionsort()

{

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

int i, key, j;

for (int index = 1; index < n; index++) {

key = arr[index];

j = index - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

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

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

cout << "Time taken by insertion sort: "

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

}

template <typename T>

void Vector<T>::quicksort()

{

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

qs(0, (\*this).n - 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 <typename T>

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

{

int i = first, j = last;

T tmp, x = arr[(first + last) / 2];

do {

while (arr[i] < x)

i++;

while (arr[j] > x)

j--;

if (i <= j)

{

if (i < j)

{

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}

i++;

j--;

}

} while (i <= j);

if (i < last)

qs(i, last);

if (first < j)

qs(first, j);

}

template <typename T>

Vector<T> Vector<T>::operator+ (const Vector<T>& other)

{

if (n != other.GetN() || n != other.GetN())

throw "Vectors aren't equal";

Vector<T> result(this->n);

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

{

result[i] = this->arr[i] + other[i];

}

return result;

}

template <class T>

Vector<T> Vector<T>::operator- (const Vector<T>& other)

{

if (n != other.GetN() || n != other.GetN())

throw "Vectors aren't equal";

Vector<T> newvector(this->n);

for (int index = 0; index < this->n; index++)

newvector[index] = this->arr[index] - other.arr[index];

return newvector;

}

template <typename T>

Vector<T> Vector<T>::operator\* (const Vector& other)

{

//if (n != other.n)

// throw "wrong n";

Vector<T> newvector(this->n);

for (int index = 0; index < this->n; index++)

newvector[index] = this->arr[index] \* other.arr[index];

return newvector;

}

template <typename T>

void Vector<T>::bubblesort()

{

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

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

{

for (int j = i + 1; j < n; j++)

{

if (arr[i] > arr[j])

{

T element;

element = arr[i];

arr[i] = arr[j];

arr[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<typename T>

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

{

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

if (arr[i] == a)

return i;

return -1;

}

template<typename T>

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

{

int k = 0;

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

if (arr[i] == a)

{

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

k = 1;

}

if (k == 0)

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

}

Main.cpp

#include "Vector.h"

#include "Matrix.h"

#include <iostream>

using namespace std;

int main()

{

srand(time(NULL));

Vector<double> v1(5);

v1[2] = 42;

Vector<double> v2(v1);

Vector <double> v3;

v1 + v2;

Matrix <double> m1(2, 2);

Matrix<double> m2(2, 2);

Matrix<double> M;

M = m1 + m2;

int t = 10;

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

{

cout << "t = " << t << endl;

Vector<double> v1(t);

v1.bubblesort();

t = t \* 10;

}

t = 10;

cout << endl;

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

{

cout << "t = " << t << endl;

Vector<double> v1(t);

v1.insertionsort();

t = t \* 10;

}

t = 10;

cout << endl;

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

{

cout << "t = " << t << endl;

Vector<double> v1(t);

v1.quicksort();

t = t \* 10;

}

t = 10;

cout << endl;

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

{

cout << "t = " << t << endl;

Matrix<int> m1(t, t);

m1 + m1;

t = t \* 10;

}

t = 1;

cout << endl;

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

{

cout << "t = " << t << endl;

Matrix<int> m1(t, t);

m1\* m1;

t = t \* 10;

}

return 0;

}