**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**

Выполнила:

студентка ИИТММ

группы 381906-2

Сорокина Е.В.

Проверил:

ассистент каф. математического обеспечения и суперкомпьютерных технологий

Лебедев И.Г.

Нижний Новгород

2020 г.

**Содержание**

[Введение 3](#_Toc39486499)

[1. Постановка задачи 4](#_Toc39486500)

[2.Руководство пользователя 5](#_Toc39486501)

[3. Руководство программиста 6](#_Toc39486502)

[3.1. Описание структуры программы 6](#_Toc39486503)

[3.2. Описание структур данных 6](#_Toc39486504)

[3.3. Описание алгоритмов 8](#_Toc39486505)

[4. Эксперименты 15](#_Toc39486506)

[Заключение 19](#_Toc39486507)

[Литература 20](#_Toc39486508)

[Приложение 21](#_Toc39486509)

# Введение

В программировании очень большую роль играет математика. Не зная элементарных математических объектов и теорем, программист не сможет решить даже самые простые задачи. В этой работе я хочу рассмотреть такие математические понятия как векторы и матрицы.

Матрица — это математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых, действительных или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов задает размер матрицы.

Если матрица состоит только из одного столбца или строки, то такой объект называется вектором. Вектор — направленный отрезок прямой, то есть отрезок, для которого указано, какая из его граничных точек является началом, а какая — концом.

В программировании вектор – это структура данных, которая является моделью динамического массива.

Для того чтобы работать с векторами и матрицами в программировании мы должны знать, что такое классы и как с ними работать.

Класс — в объектно-ориентированном программировании, представляет собой шаблон для создания объектов, обеспечивающий начальные значения состояний: инициализация полей-переменных и реализация поведения функций или методов.

Для выполнения моей задачи мне также понадобится знания по теме шаблоны функций и классов.

Шаблоны функций — это инструкции, согласно которым создаются локальные версии шаблонной функции для определенного набора параметров и типов данных, это описание поведения функций, которые могут вызываться для объектов разных типов. Другими словами, шаблон функции 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. Шаблон класса позволяет задать тип для объектов, используемых в классе. Но прежде чем перейти к определению шаблона класса, рассмотрим проблему, с которой мы можем столкнуться и которую позволяют решить шаблоны.

В этой работе я продемонстрирую реализацию классов для работы с матрицами и векторами с использованием шаблонов.

# 1. Постановка задачи

1. Написать классы для работы с векторами и матрицами, используя шаблоны.

2. Реализовать сортировки: пузырьком, вставкой, быстрая;

3. Продемонстрировать работу векторов 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 +, -, \*, / для векторов (вектор +, -, \*, / вектор), матриц (матрица +, -, \* матрица), матрично-векторные (матрица \* вектор и наоборот);
* в классе вектор должна быть возможность отсортировать его тремя способами.

5. Сравнить время работы и сделать вывод.

# 2.Руководство пользователя

Использование программы пользователем:

1. Запустить окно программы Vector.exe из папки Debug;
2. Создать объект типа Vector или Matrix;
3. Расширить матрицу или вектор на определенное количество элементов, используя конструкторы или методы доступа к защищенным полям;
4. Произвести необходимые математические операции.

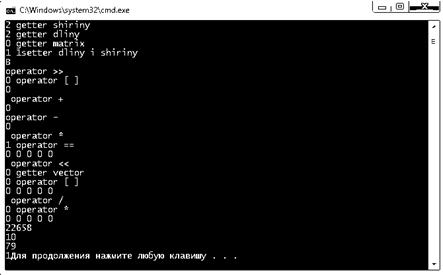


Рисунок 1. Интерфейс программы(скриншот).

# 3. Руководство программиста

# 3.1. Описание структуры программы

Программа состоит из одного решения, называющегося Vector\_and\_Matrix.

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

* В модуле Matrix.h определен класс Matrix, а также объявлены все его методы;
* В модуль Matrix\_pro.h вынесены все определения методов класса Matrix;
* В модуле Vector.h определен класс Vector, являющийся наследником класса Matrix;
* В модуль Vector\_pro.h вынесены все определения методов класса Vector;
* В модуле main.cpp определена стандартная функция int main(), внутри которой содержится набор действий с объектами типа Matrix и Vector.

# 3.2. Описание структур данных

В программе определены следующие шаблонные классы: Class Matrix и Class Vector.

Внутри класса Matrix определены следующие поля:

* Type\*\* matrix – шаблонный двойной указатель;
* int width – целочисленный размер ширины матрицы;
* int length – целочисленный размер длины 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(int, int) – конструктор инциализатор, принимает на вход два параметра типа int, создает динамический массив заданных размеров, заполняет его 0, поля width и length становятся значениями переданными в конструктор соответственно;
* Matrix(const Matrix& lhs) – конструктор копирования, принимает на вход объект типа Matrix, создает объект с теми же характеристиками, что и переданный, и матрицей того же содержимого;
* Virtual ~Matrix() – виртуальный деструктор, очищает выделенную методами и конструкторами память;
* int get\_width() – метод, возвращающий ширину матрицы;
* int get\_length() – метод, возвращающий длину матрицы;
* Type\*\* get\_array() – метод, возвращающий массив, содержащий в себе все элементы матрицы;
* void set\_width\_length(int,int) – 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 хватило места, останутся в матрице. В случае если матрица имеет любую размерность, равную 0, метод ведет себя аналогично конструктору-инциализатору;
* Type\*& operator [ ](int) - перегрузка оператора индексации, позволяет возвращать элемент матрицы по индексу;
* friend Matrix operator +(Matrix & lhs, Matrix & rhs) – перегрузка оператора суммы, позволяет возвращать матрицу, являющуюся суммой двух матриц;
* friend Matrix operator –(Matrix& lhs, Matrix& rhs) – перегрузка оператора ¬– , аналогична в своей сути перегрузке оператора +;
* Matrix& operator =(const Matrix & lhs) - перегрузка оператора =, позволяет присваивать один объект типа Matrix другому;
* friend bool operator ==(Matrix& lhs, Matrix& rhs) – перегрузка оператора сравнения, позволяет сравнивать объекты типа Matrix;
* friend Matrix operator \*(const Matrix& lhs, const Matrix& rhs) - перегрузка оператора умножить, позволяет перемножать обьекты типа Matrix;
* friend ostream& operator <<(ostream& out, const Matrix& rhs) – перегрузка оператора вывода, позволяет выводить матрицы на экран;
* friend istream& operator >>(istream& in, Matrix& rhs) - перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера либо задавать размер и вводить самостоятельно.

Единственным «Private» методом является метод

* void MemorySize(int, int) –выделяет память до размеров матрицы, переданных в функцию;
* Шаблонный класс Vector является public, наследником класса Matrix, поэтому никаких полей в этом классе не определено. Также почти 70% методов, он наследует от класса-родителя, поэтому внутри него, определены только специфичные для него методы:
* T1\* get\_vector() – метод, возвращающий массив, содержащий в себе все элементы объекта Vector;
* T1 operator [](const int) - перегрузка оператора индексации, позволяет возвращать элемент вектора по индексу;
* void set\_length(int) – метод, принимающий новые размеры вектора, приводящий вектор к этим размерам. В случае если новый вектор больше старого, новые элементы равняются нулю. В противном случае, все элементы, на которые хватило места, останутся в векторе. В случае если вектор имеет размерность равную 0, метод ведет себя аналогично конструктору инциализатору;
* friend Vector operator /(Vector& lhs, Vector& rhs) – перегрузка оператора / возвращает вектор, координаты которого являются частным от деления двух соответствующих координат операндов. В случае если делитель равен нулю, значение частного считается равным делимому;
* friend Type operator \*(Vector<Type>& lhs, Vector & rhs) – перегрузка оператора умножить, возвращает скалярное произведение операндов;
* friend istream& operator >>(istream& in, Vector& rhs) - перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера либо задавать размер и вводить самостоятельно;
* clock\_t BubbleSort() - сортировка методом «Пузырек». Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее;
* clock\_t InsertionSort() - сортировка методом «Вставка». Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее.
* clock\_t QuickSort() – сортировка методом «Быстрая».

Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее.

# 3.3. Описание алгоритмов

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.

Блок-схемы алгоритмов сортировки:

1. Пузырьком

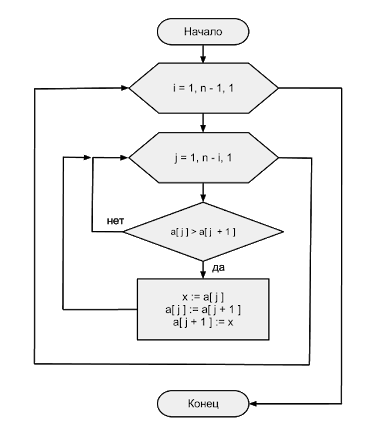


Рисунок 2. Блок-схема алгоритма сортировки пузырьком.

1. Вставками

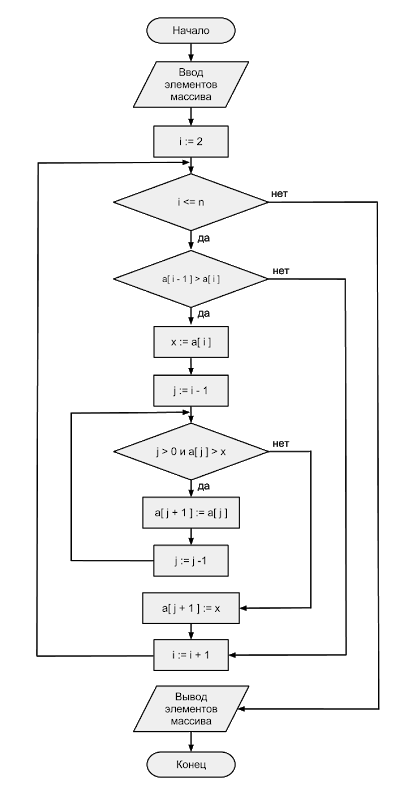


Рисунок 3. Блок-схема алгоритма сортировки вставками.

1. Быстрая

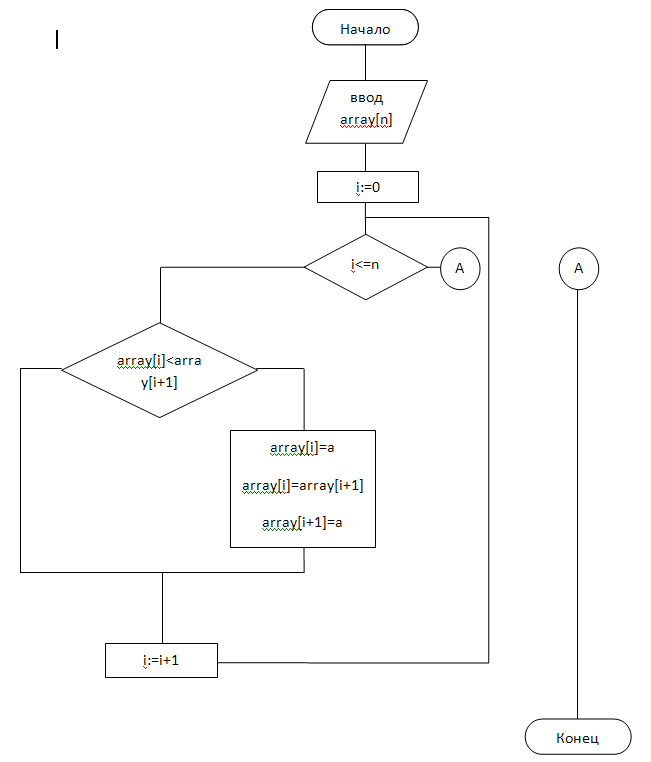


Рисунок 4. Блок-схема алгоритма быстрой сортировки.

Блок-схемы алгоритмов операций над матрицами:

1. Алгоритм суммирования или вычитания матриц:

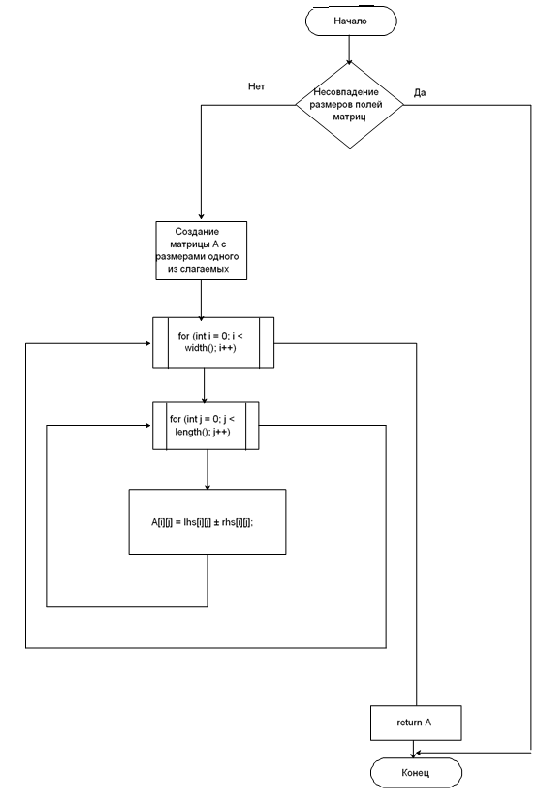


Рисунок 5: Блок-схема алгоритма сложения/вычитания матриц.

1. Алгоритм умножения матрицы на матрицу:

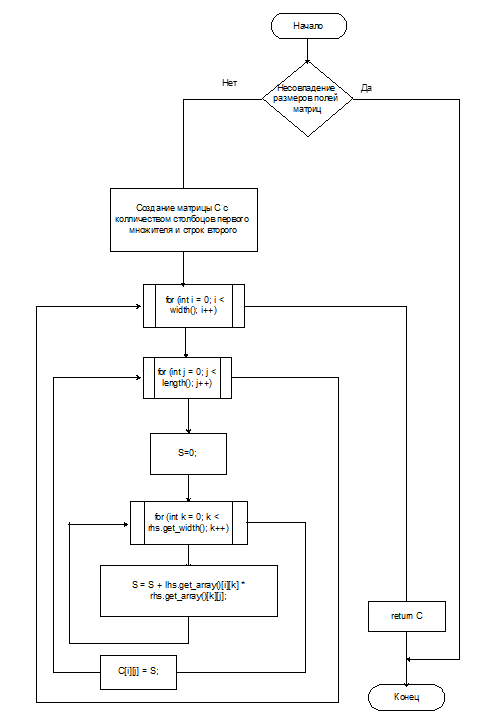


Рисунок 6: Блок-схема алгоритма умножения матриц.

Поскольку класс Vector - это публичный наследник класса Matrix, то для него нет смысла описывать каким-либо образом арифметическую операцию сложения, так как эта операция в числе других наследуется от класса Matrix.

1. Алгоритм деления двух векторов:

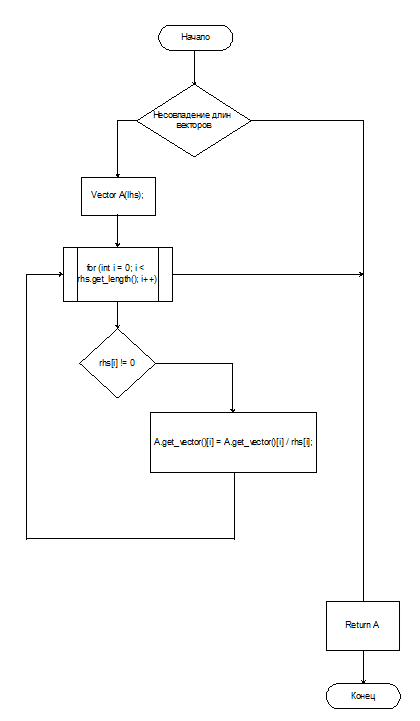


Рисунок 7: Блок-схема умножения одной матрицы на другую.

1. Алгоритм умножения двух векторов:

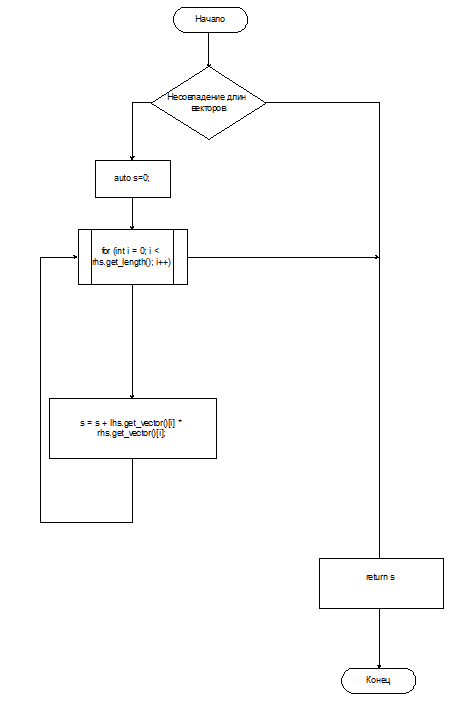


Рисунок 8: Блок-схема умножения двух векторов.

# 

# 4. Эксперименты

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

Matrix<Type> A(lhs.get\_width(), rhs.get\_length());

for (int i = 0; i < rhs.get\_width(); i++)

{

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

{

A[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):

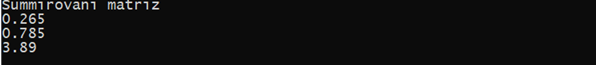


Рисунок 9: Замеры времени, полученные при выполнении сложения матриц (сек.).

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

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 0.265 |
| 2000 | 0.785 |
| 4000 | 3.89 |

Таблица 1: Результаты замеров времени суммирования матриц.

Заметим, что при увеличении количества элементов в 2 раза относительно предыдущего, время увеличивается примерно 4 раза.

Теперь аналогично оценим асимптотическую сложность матричного умножения:

Matrix<F> C(lhs.get\_width(), rhs.get\_length());

F S;

for (int i = 0; i < lhs.get\_width(); i++)

{

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

{

S = 0;

for (int k = 0; k < rhs.get\_width(); k++)

{

S = S + lhs.get\_array()[i][k] \* rhs.get\_array()[k][j];

}

C[i][j] = S;

}

}

Асимптотическая сложность умножения:

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

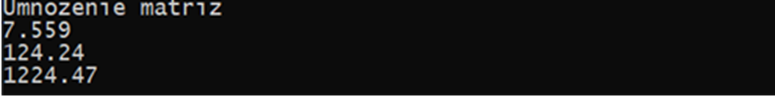


Рисунок 10: Замеры времени, полученные при выполнении умножения матриц (сек.).

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

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 7.559 |
| 2000 | 124.24 |
| 4000 | 1224.47 |

Таблица 2: Результаты замеров времени умножения матриц.

Заметим, что соотношение времени чуть выше кубического. Однако результаты близки к ожидаемым. Теперь аналогично оценим асимптотическую сложность векторно-матричного умножения:

Matrix<F> C(lhs.get\_width(), rhs.get\_length());

F S;

for (int i = 0; i < lhs.get\_width(); i++)

{

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

{

S = 0;

for (int k = 0; k < rhs.get\_width(); k++)

{

S = S + lhs.get\_array()[i][k] \* rhs.get\_array()[k][j];

}

C[i][j] = S;

}

}

Асимптотическая сложность умножения:

Отметим, что важной для асимптотической сложности является только длина вектора (равная ширине матрицы).

Теперь произведем замеры времени умножения вектора на матрицу:



Рисунок 11: Замеры времени, полученные при выполнении умножения вектора на матрицу(сек.).

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

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 0.003 |
| 2000 | 0.023 |
| 4000 | 0.11 |

Таблица 3: Результаты замеров времени умножения матриц.

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



Рисунок 12: Замеры времени, полученные при выполнении сортировки вектора (сек.)



Рисунок 13: Замеры времени, полученные при выполнении сортировки вектора(сек.)



Рисунок 14: Замеры времени, полученные при выполнении сортировки вектора(сек.)

Полученные мной данные занесем в таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Вид сортировки | Асимптотическая сложность | Количество элементов | Время (сек.) |
| BubbleSort |  | 100000 | 12.608 |
| 200000 | 50.296 |
| 400000 | 200.226 |
| InsertionSort |  | 100000 | 1.786 |
| 200000 | 6.981 |
| 400000 | 28.339 |
| QuickSort |  | 100000 | 0.012 |
| 200000 | 0.022 |
| 400000 | 0.042 |

Таблица 4: Результаты сортировок по времени для массивов

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

# Заключение

Итак, в данной работе было продемонстрировано действие программы с использованием шаблонов, которая реализует классы для работы с векторами и матрицами.

Эксперименты показывают, что с некоторыми погрешностями, затраченное на реализацию матричных, матрично-векторных операций и сортировок время совпадает с ожидаемым относительно асимптотической сложности временем.

# Литература

1. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — С. 406.
2. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003.
3. Официальный сайт Habr. – Режим доступа <https://habr.com/ru/post/339656/>
4. Официальный сайт Microsoft. – Режим доступа <https://docs.microsoft.com/ru-ru/cpp/c-language/main-function-and-program-execution?view=vs-2019>

# Приложение

Код программы:

**Vector.h**

#pragma once

#include "Matrix.h"

template <class T1>

class Vector :public Matrix<T1>

{

public:

Vector() :Matrix() {}

Vector(int \_size) :Matrix(1, \_size) {}

Vector(const Vector& t) :Matrix(t) {}

~Vector() {}

T1\* get\_vector();

using Matrix:: operator =;

T1& operator [](const int i);

void set\_length(int \_resize);

template<class J>

friend Vector<J> operator /(Vector<J>& lhs, Vector<J>& rhs);

template<class Y>

friend Y operator \*(Vector<Y>& lhs, Vector<Y>& rhs);

template<class K>

friend istream& operator >>(istream& in, Vector<K>& rhs);

friend void In\_Vector(Vector& lhs) {

for (int i = 0; i < lhs.get\_length(); i++)

lhs[i] = rand();

}

template<class P>

friend clock\_t BubbleSort(Vector<P>& A);

template<class Q>

friend clock\_t InsertionSort(Vector<Q>& a);

template<class K>

friend clock\_t QuickSort(Vector<K>& p, int first, int last);

};

#include "Vector\_pro.h"

**Vector\_pro.h**

#pragma once

template <class T1>

T1\* Vector<T1>::get\_vector()

{

return get\_array()[0];

}

template<class T2>

T2& Vector<T2>::operator [](const int i)

{

return get\_vector()[i];

}

template <class T1>

void Vector<T1>::set\_length(int \_resize)

{

set\_width\_length(1, \_resize);

}

template<class F>

Vector<F> operator /(Vector<F>& lhs, Vector<F>& rhs)

{

if (lhs.get\_length() != rhs.get\_length())throw length\_error("Vector's size not equal");

Vector<F> A(lhs);

for (int i = 0; i < A.get\_length(); i++)

{

if (rhs[i] != 0)

{

A.get\_vector()[i] = A.get\_vector()[i] / rhs[i];

}

}

return A;

}

template <class Y>

Y operator \*(Vector<Y>& lhs, Vector<Y>& rhs)

{

if (lhs.get\_length() != rhs.get\_length())throw length\_error("Vector's size not equal");

Y s = 0;

for (int i = 0; i < rhs.get\_length(); i++)s = s + lhs.get\_vector()[i] \* rhs.get\_vector()[i];

return s;

}

template<class U>

istream& operator >>(istream& in, Vector<U>& rhs)

{

if (rhs.get\_length() == 0)

{

int \_size;

in >> \_size;

rhs.set\_length(\_size);

}

for (int i = 0; i < rhs.get\_length(); i++)

{

in >> rhs.get\_vector()[i];

}

return in;

}

template<class P>

clock\_t BubbleSort(Vector<P>& A)

{

clock\_t t1, t2;

t1 = clock();

P tmp = 0;

int i, j = 0;

t1 = clock();

for (int i = 0; i < A.get\_length(); i++)

{

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

{

if (A.get\_vector()[j] > A.get\_vector()[j + 1])

{

tmp = A.get\_vector()[j];

A.get\_vector()[j] = A.get\_vector()[j + 1];

A.get\_vector()[j + 1] = tmp;

}

}

}

t2 = clock();

return (t2 - t1);

}

template<class Q>

clock\_t InsertionSort(Vector<Q>& a)

{

clock\_t t1, t2;

t1 = clock();

Q newElement;

int location;

for (int i = 1; i < a.get\_length(); i++)

{

newElement = a.get\_vector()[i];

location = i - 1;

while (location >= 0 && a.get\_vector()[location] > newElement)

{

a.get\_vector()[location + 1] = a.get\_vector()[location];

location = location - 1;

}

a.get\_vector()[location + 1] = newElement;

}

t2 = clock();

return t2 - t1;

}

template<class K>

clock\_t QuickSort(Vector<K>& p, int first, int last)

{

clock\_t t1, t2, t3;

t1 = clock();

int i = first, j = last;

K tmp;

K x = p.get\_vector()[(first + last) / 2];

do {

while (p.get\_vector()[i] < x)

i++;

while (p.get\_vector()[j] > x)

j--;

if (i <= j)

{

if (i < j)

{

tmp = p.get\_vector()[i];

p.get\_vector()[i] = p.get\_vector()[j];

p.get\_vector()[j] = tmp;

}

i++;

j--;

}

} while (i <= j);

if (i < last)

t3 = QuickSort(p, i, last);

if (first < j)

t3 = QuickSort(p, first, j);

t2 = clock();

return (t2 - t1);

}

**Matrix.h**

#pragma once

#include <iostream>

using namespace std;

template <class Type>

class Matrix

{

public:

Matrix() :width(0), length(0), matrix(0) {}

Matrix(int \_a, int \_b);

Matrix(const Matrix& lhs);

virtual ~Matrix();

int get\_width()const;

int get\_length()const;

Type\*\* get\_array();

void set\_width\_length(int \_width, int \_length);

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

template<class V>

friend Matrix<V> operator +(Matrix<V>& lhs, Matrix<V>& rhs);

template <class A>

friend Matrix<A> operator -(Matrix<A>& lhs, Matrix<A>& rhs);

Matrix& operator =(Matrix& lhs);

template <class C>

friend bool operator ==(Matrix<C>& lhs, Matrix<C>& rhs);

template <class D>

friend Matrix<D> operator \*(Matrix<D>& lhs, Matrix<D>& rhs);

template <class E>

friend ostream& operator <<(ostream& out, Matrix<E>& rhs);

template<class W>

friend istream& operator >>(istream& in, Matrix<W>& rhs);

private:

void MemorySize(int \_width, int \_length)

{

for (int i = 0; i < \_width; i++) matrix[i] = new Type[\_length];

}

Type\*\* matrix;

int width;

int length;

};

#include "Matrix\_pro.h";

**Matrix\_proc.h**

#pragma once

template<class A1>

Matrix<A1>::Matrix(int \_a, int \_b)

{

if ((\_a < 0) || (\_b < 0))

{

if (\_a < 0) throw length\_error("uncorrect width");

if (\_b < 0) throw length\_error("uncorrect length");

}

matrix = new A1 \* [\_a];

MemorySize(\_a, \_b);

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

{

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

{

matrix[i][j] = 0;

}

}

width = \_a;

length = \_b;

}

template <class A2>

Matrix<A2>::Matrix(const Matrix& lhs)

{

matrix = new A2 \* [lhs.width];

MemorySize(lhs.width, lhs.length);

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

{

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

{

matrix[i][j] = lhs.matrix[i][j];

}

}

width = lhs.width;

length = lhs.length;

}

template <class A3>

Matrix<A3>::~Matrix()

{

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

delete[] matrix;

}

template<class A1>

int Matrix<A1>::get\_width() const

{

return width;

}

template <class A2>

int Matrix<A2>::get\_length() const

{

return length;

}

template <class A3>

A3\*\* Matrix<A3>::get\_array()

{

return matrix;

}

template<class A1>

void Matrix<A1>::set\_width\_length(int \_width, int \_length) {

if ((\_width < 0) || (\_length < 0))

{

if (\_width < 0) throw length\_error("uncorrect width");

if (\_length < 0) throw length\_error("uncorrect length");

}

Matrix<A1>D(width, length);

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

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

D.get\_array()[i][j] = (\*this).get\_array()[i][j];

}

}

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

delete[] matrix;

width = \_width;

length = \_length;

matrix = new A1 \* [width];

for (int i = 0; i < width; i++) matrix[i] = new A1[length];

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

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

matrix[i][j] = 0;

}

}

int wb = (D.get\_width() < width) ? D.get\_width() : width;

int lb = (D.get\_length() < length) ? D.get\_length() : length;

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

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

matrix[i][j] = D.get\_array()[i][j];

}

}

}

template <class A3>

A3\*& Matrix<A3>::operator[](const int i)

{

if ((i > width) || (i > length) || (i < 0))throw length\_error("uncorrect index");

return matrix[i];

}

template<class A1>

Matrix<A1>& Matrix<A1>::operator =(Matrix& lhs) {

if (this == &lhs) {

return (\*this);

}

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

delete[] matrix;

width = lhs.get\_width();

length = lhs.get\_length();

matrix = new A1 \* [width];

MemorySize(width, length);

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

{

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

{

matrix[i][j] = lhs[i][j];

}

}

return \*this;

}

template <class Type>

Matrix<Type> operator +(Matrix<Type>& lhs, Matrix<Type>& rhs) {

if ((lhs.get\_width() != rhs.get\_width()) || (lhs.get\_length() != rhs.get\_length()))throw length\_error("matrix size not equal");

Matrix<Type> A(lhs.get\_width(), rhs.get\_length());

for (int i = 0; i < rhs.get\_width(); i++)

{

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

{

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

}

}

return A;

}

template <class Type>

Matrix<Type> operator -(Matrix<Type>& lhs, Matrix<Type>& rhs) {

if ((lhs.get\_width() != rhs.get\_width()) || (lhs.get\_length() != rhs.get\_length()))throw length\_error("matrix size not equal");

Matrix<Type> A(lhs);

for (int i = 0; i < rhs.get\_width(); i++)

{

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

{

A[i][j] = A[i][j] - rhs[i][j];

}

}

return A;

}

template <class E>

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

{

if ((lhs.get\_width() != rhs.get\_width()) || (lhs.get\_length() != rhs.get\_length()))

{

return false;

}

else {

for (int i = 0; i < lhs.get\_width(); i++)

{

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

{

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

}

}

}

return true;

}

template <class F>

Matrix<F> operator \*(Matrix<F>& lhs, Matrix<F>& rhs) {

if (lhs.get\_length() != rhs.get\_width())throw length\_error("matrix size uncorrect");

Matrix<F> C(lhs.get\_width(), rhs.get\_length());

F S;

for (int i = 0; i < lhs.get\_width(); i++)

{

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

{

S = 0;

for (int k = 0; k < rhs.get\_width(); k++)

{

S = S + lhs.get\_array()[i][k] \* rhs.get\_array()[k][j];

}

C[i][j] = S;

}

}

return C;

}

template <class R>

ostream& operator <<(ostream& out, Matrix<R>& rhs)

{

for (int i = 0; i < rhs.get\_width(); i++)

{

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

{

out << rhs.get\_array()[i][j] << " ";

}

out << endl;

}

return out;

}

template <class K>

istream& operator >>(istream& in, Matrix<K>& rhs)

{

if ((rhs.get\_length() == 0) || (rhs.get\_width() == 0))

{

cout << "Vvedite width&Length" << endl;

int a, b;

in >> a >> b;

rhs.set\_width\_length(a, b);

}

for (int i = 0; i < rhs.get\_width(); i++)

{

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

{

in >> rhs[i][j];

}

}

return in;

}

**main.cpp**

#include <cstdlib>

#include <iostream>

#include <malloc.h>

#include <time.h>

#include "Matrix.h"

#include "Vector.h"

#include <vector>

using namespace std;

int main() {

try {

/\*Matrix<int>B;

Matrix<int>A(2, 2);

Matrix<int>C(2,2);

//---------------------------------------------------------------------------

cout << C.get\_width() << " getter shiriny\n";

cout << C.get\_length() << " getter dliny\n";

cout << C.get\_array()[0][0] << " getter matrix\n";

//---------------------------------------------------------------------------

B.set\_width\_length(1, 1);

A.set\_width\_length(1, 1);

cout << B.get\_width() << " " << B.get\_length() << "setter dliny i shiriny\n";

//---------------------------------------------------------------------------

//cin >> A >> B;

cout << "operator >> \n";

cout << B[0][0] << " operator [ ]\n";

//---------------------------------------------------------------------------

C = A + B;

cout << C << " operator +\n";

C = A - B;

cout << C << "operator -\n";

cout << A\*B << " operator \*\n";

cout << (A == B) << " operator ==\n";

//---------------------------------------------------------------------------

Vector<float>G(5);

Vector<float>H(5);

Vector<float>I(H);

A.set\_width\_length(3, 3);

Vector<float> L(3);

Matrix<float>s(3, 3);

//---------------------------------------------------------------------------

cout << H << " operator <<\n";

cout << H.get\_vector()[0] << " getter vector\n";

cout << H[0] << " operator [ ]\n";

G.set\_length(5);

//---------------------------------------------------------------------------

//cin >> G;

//cin >> H;

cout << H / G << " operator /\n";

I= G + H;

G.set\_length(3);

L = s\*L;

float p = G\*H;

cout << I;

Vector<int>K(10000);

In\_Vector(K);

cout << BubbleSort(K) << endl;

cout << InsertionSort(K) << endl;

cout << QuickSort(K, 0, K.get\_length() - 1) << endl;

Matrix <int> P(1, 2);

P[0][0] = 1;

cout << P[0][0];/\*/

//------------------------------------------------------------------------------

cout << "QuickSort" << endl;

vector<double> A(3);

Vector<int>Z(1000);

Matrix<int>N(1000, 1000);

int j = 0;

for (int i = 100000; j < 3; i \*= 2) {

Z.set\_length(i);

In\_Vector(Z);

clock\_t start = clock();

QuickSort(Z, 0, Z.get\_length() - 1);

clock\_t end = clock();

A[j] = (double)(end - start) / CLOCKS\_PER\_SEC;

j++;

}

for (int i = 0; i < 3; i++) {

cout << A[i] << endl;

}

}

catch (length\_error& a) {

cout << a.what() << endl;

}

return 0;

}