Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики

ОТЧЕТ  
по лабораторной работе

**Матрицы**

Выполнила студентка группы 381903-3

Клименко Ксения Евгеньевна

Проверил Лебедев И.Г.

Нижний Новгород

2020 г.

**Оглавление**

[**Введение** 3](#_Toc59132061)

[**1.** **Постановка учебно-практической задачи** 4](#_Toc59132062)

[**1.1** **Основные понятия и определения** 4](#_Toc59132063)

[**1.2 Операции над векторами** 4](#_Toc59132064)

[**1.3 Операции над матрицами** 4](#_Toc59132065)

[**2. Руководство пользователя** 6](#_Toc59132066)

[**3. Руководство программиста** 7](#_Toc59132067)

[**3.1 Описание структуры программы** 7](#_Toc59132068)

[**3.2 Описание структур данных** 7](#_Toc59132069)

[**3.3 Описание алгоритмов** 8](#_Toc59132070)

[**3.4 Описание структуры программы Vector** 9](#_Toc59132071)

[**3.5 Описание структуры программы Matrix** 12](#_Toc59132072)

[**Эксперименты** 15](#_Toc59132073)

[**Заключение** 16](#_Toc59132074)

[**Список литературы** 17](#_Toc59132075)

[**Приложение** 18](#_Toc59132076)

# **Введение**

Понятие Матрица в европейской науке было введено в работах У. Гамильтона и А. Кэли в середине XIX века.

Матричные обозначения широко распространены в современной математике и её приложениях. Матрица – полезный аппарат для исследования многих задач теоретической и прикладной математики. Так, одной из важнейших является задача нахождения решения систем линейных алгебраических уравнений.

Следствием разнообразия областей применения матричного аппарата в современной науке является наличие в любом из больших математических программных комплексов подсистем, выполняющих операции над матрицами, а также существование специальных программных библиотек, рассчитанных на обработку огромных матриц, в том числе с использованием распределенных вычислений.

Помимо матриц общего вида, для которых наиболее естественной и наиболее часто используемой представляется программная реализация в виде двумерного массива, в математических приложениях выделяются различные матрицы специальных видов. Для таких матриц предпочтительно создание собственных способов хранения и обработки, учитывающих специфику их структуры, и потому более эффективных.

Изучению некоторых из них посвящена данная работа.

# **Постановка учебно-практической задачи**

### **Основные понятия и определения**

Матрица – прямоугольная таблица каких-либо элементов aij, состоящая из m строк и n столбцов:

a11 … a1n

A= … … …

am1 … amn

Над матрицей можно производить действия по правилам матричной алгебры. При m=n матрица называется квадратной, а число n – её порядком. Набор элементов матрицы (a11, a22, …, ann) называется главной диагональю.

Определение матрицы возможно также через понятие вектора. Вектор – набор ai, состоящий из n элементов. Тогда матрица из m строк и n столбцов может быть определена как вектор из n элементов, где каждый элемент, в свою очередь, является вектором из m элементов.

### **1.2 Операции над векторами**

Пусть заданы два вектора A = (a1, a2, …, an) и B = (b1, b2, …, bn). Рассмотрим следующие основные операции над векторами:

• Сравнение (A = B). Вектора считаются равными тогда и только тогда, когда ai = bi при всех i = 1..n. • Прибавление скаляра (A + a). Результатом сложения вектора A и скаляра ‘а’ называется вектор A’ = (a1 + a, a2 + a, …, an + a).

• Вычитание скаляра (A – a). Результатом вычитания вектора A и скаляра ‘а’ называется вектор A’ = (a1 – a, a2 – a, …, an – a).

• Умножение на скаляр (A \* a). Результатом умножения вектора A на скаляр ‘а’ называется вектор A’ = (a1 \* a, a2 \* a, …, an \* a).

• Сложение векторов (A + B). Результатом сложения векторов A и B называется вектор С = (a1 + b1, a2 + b2, …, an + bn).

• Вычитание векторов (A – B). Результатом вычитания векторов A и B называется вектор С = (a1 – b1, a2 – b2, …, an – bn).

• Скалярное произведение векторов (A \* B). Скалярным произведением векторов A и B называется скалярная величина с =√(∑ai\*bi).

### **1.3 Операции над матрицами**

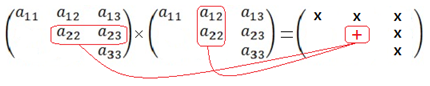
Пусть заданы две матрицы A = (aii) и B = (bii) и вектор V = (vi), где i = 1...n. Рассмотрим следующие основные операции над матрицами:

• Сравнение (A = B). Матрицы считаются равными тогда и только тогда, когда aii = bii при всех i = 1…n.

• Умножение на скаляр (A \* d). Результатом умножения матрицы A на скаляр ‘d’ называется матрица D = (dii), где dii = aii \* d, при всех i = 1…n.

• Сложение матриц (A + B). Результатом сложения матриц A и B называется матрица C = (cii), где cii = aii + bii при всех i = 1…n.

• Вычитание матриц (A – B). Результатом вычитания матриц A и B называется матрица C = (cii), где cii = aii - bii при всех i = 1…n.

• Произведением матриц A и B называется такая матрица С, элементы которой находятся по такому правилу: сij равен сумме произведений элементов i-той строки матрицы A на соответствующие элементы j-го столбца матрицы B. Таким образом, умножение осуществляется по правилу умножения строки на столбец.

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих эффективное хранение квадратных матриц и выполнение основных операций над ними:

• сложение/вычитание;

• умножение;

• сравнение.

Программные средства должны содержать:

• класс Вектор;

• класс Матрица.

В классах должны быть:

1)конструкторы (по умолчанию, инициализатор, копирования), деструктор, доступ к защищенным полям;

2)перегруженные операции: +, -, \*, =, ==, !=, [ ], потоковый ввод и вывод;

3)перегруженные операции +, -, \* должны быть реализованы для векторов (вектор +-\* вектор), (вектор +-\* скаляр), матриц (матрица +-\* матрица).

# **2. Руководство пользователя**

1. Создайте переменную типа TVektor<T> или TMatrix<T>, где вместо T нужно написать используемый далее тип, например: int, double, float и т.д.
2. Чтобы использовать какую-либо доступную функцию для определенной переменной, записывайте ее через точку после вашей переменной.
3. Пользуясь операторами и функциями, вы можете написать программу для осуществления операций с векторами и матрицами.

# **3. Руководство программиста**

## **3.1 Описание структуры программы**

В данной программе 6 модулей: main.cpp, TMatrix.h, TVector.h, test\_main.cpp, test\_Matrix.cpp, test\_Vector.cpp написаны

* В модуле main.cpp определена стандартная функция int main(), в которой пользователю предлагается написать программу для осуществления операций с объектами типа TMatrix и TVector.
* В модуле TMatrix.h определен класс Matrix, в котором объявлены и определены его методы.
* В модуле TVector.h определен класс Vector, в котором объявлены и определены его методы.
* В модулях test\_main.cpp, test\_Matrix.cpp, test\_Vector.cpp написаны тесты на основе модулей main.cpp, Vector.h, Matrix.h соответственно

## **3.2 Описание структур данных**

Структура данных - модель данных в виде математической структуры S = (M1, …, Mk, p1,…,pn), где M1, …, Mk – базисные множества, p1,…,pn – отношения между элементами базисных множеств.

В соответствии с этим можно привести следующие определения:

1. Структура данных Вектора V = (v1, v2, …, vn) есть

Sv = (Mv pv), где

Mv = {v1,v2, …, vn} – базисное множество,

и, j=i+1  
pa{vi,vj}= л, j!=i+1 - отношение следования.

1. Структура данных Матрицы A = (aij), где i = 1..n; j = 1..n есть

Sa = (Ma, p1a, p2a), где

Ma = {a11, a12, …, ann-1,ann} – базисное множество,

и, j=i+1   
 p1a{aik,ajk}= k=1...n  
 л, j!=i+1   
 - отношение следования.  
 и, j=i+1   
 p2a{aik,ajk}= k=1...n  
 л, j!=i+1

1. При определении Матрицы через Вектор (A = (vi), где vi – вектор из n элементов) структура данных примет вид:

S1a = (M1a, p3a), где

M1a = {v1, v2, …, vn} – базисное множество,

и, j=i+1  
p3a{vi,vj}= - отношения следования.  
 л, j!=i+1



Очевидно сходство в задании структуры данных Вектор, как набора элементов, связанных отношением следования, и структуры данных Матрица, как набора элементов - векторов, связанных отношением следования. Этот факт позволяет единообразно организовать алгоритмы обработки векторов и матриц, а, следовательно, использовать при разработке требуемых классов механизм наследования.

## **3.3 Описание алгоритмов**

**3.2.1 Вектор**

Для структуры данных Вектор предлагается реализовать следующие операции:

• сравнение;

• прибавление/вычитание скаляра;

• умножение на скаляр;

• сложение/вычитание векторов;

• скалярное произведение векторов;

**3.2.2 Матрица**

Для структуры данных Матрица предлагается реализовать следующие операции:

• сравнение:

• сложение/вычитание матриц:

• умножение матриц.

Как и в случае с вектором операции для структуры данных Матрица опираются на формулы, приведенные в разделе 2.

## **3.4 Описание структуры программы Vector**

Для реализации алгоритмов будет реализовано 2 класса:

* Класс Vector для векторов
* Класс Matrix, который будет использовать класс векторов.

Хранение матрицы осуществлено в виде набора векторов разной длины:

Векторы

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Матрица

|  |
| --- |
|  |
|  |
|  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
|  |  |

**3.4.1 GetSize**

int GetSize()

{ return Size; }

Для функции с возвращающим значением типа int по получению размера в качестве параметров приходит пустой список параметров. Функция возвращает Size (размер).

**3.4.2 GetStartIndex**

int GetStartIndex()

{ return StartIndex; }

Для функции с возвращающим значением типа int по получению стартового индекса в качестве параметров приходит пустой список параметров. Функция возвращает StartIndex (стартовый индекс).

**3.4.3 operator [ ]**

template <class ValType>

ValType& TVector<ValType>::operator[](int pos)

{ return pVector[pos - StartIndex]; }

Для функции типа класса по определении оператора [ ] в качестве параметров приходит нужная позиция. Функция возвращает pVector от разности pos (нужная позиция) и StartIndex (стартовый индекс).

**3.4.4 operator = =**

template <class ValType>

bool TVector<ValType>::operator==(const TVector &v)

{ //тело функции представлено в приложении }

Для функции с возвращающим значением типа bool по сравнению на равенство двух векторов в качестве параметров приходит постоянная ссылка на объект.

**3.4.5 operator** !=

template <class ValType>

bool TVector<ValType>::operator!=(const TVector &v)

{ //тело функции представлено в приложении }

Для функции с возвращающим значением типа bool по сравнению на неравенство двух векторов в качестве параметров приходит постоянная ссылка на объект.

**3.4.6 operator =**

template <class ValType>

TVector < ValType>&TVector<ValType>::operator=(const TVector &v)

{ //тело функции представлено в приложении }

Для функции типа класса по присваиванию в качестве параметров приходит постоянная ссылка на объект.

**3.4.7 operator + (int)**

template <class ValType>

TVector<ValType> TVector<ValType>::operator+(const ValType& val)

{ //тело функции представлено в приложении }

Для функции типа класса по операции сложения вектора со скаляром в качестве параметров приходит скаляр. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.8 operator - (int)**

template<typename ValType>

TVector<ValType> TVector<ValType>::operator-(const ValType& val)

{ //тело функции представлено в приложении }

Для функции типа класса по операции вычитания из вектора скаляра в качестве параметров приходит скаляр. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.9 operator \* (int)**

template <class ValType>

TVector<ValType> TVector<ValType>::operator\*(const ValType& val)

{ //тело функции представлено в приложении }

Для функции типа класса по операции умножения вектора на скаляр в качестве параметров приходит скаляр. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.10 operator +**

template <class ValType>

TVector < ValType> TVector<ValType>::operator+(const TVector <ValType> &v)

{ //тело функции представлено в приложении }

Для функции типа класса по операции сложения двух векторов в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.11 operator –**

template <class ValType>

TVector < ValType> TVector<ValType>::operator-(const TVector <ValType> &v)

{ //тело функции представлено в приложении }

Для функции типа класса по операции вычитания двух векторов в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.12 operator \***

template <class ValType>

ValType TVector<ValType>::operator\*(const TVector& v)

{ //тело функции представлено в приложении }

Для функции типа класса по операции умножения двух векторов в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.4.13 istream**

friend std::istream& operator>>(std::istream& in, TVector& v)

{

for (int i = 0; i < v.Size; i++)

in >> v.pVector[i];

return in;

}

Для функции ввода вектора приходят 2 параметра.

**3.4.14 ostream**

friend std::ostream& operator<<(std::ostream& out, const TVector& v)

{

for (int i = 0; i < v.Size; i++)

out << v.pVector[i] << ' ';

return out;

}

Для функции вывода вектора приходят 2 параметра.

## **3.5 Описание структуры программы Matrix**

**3.5.1 operator = =**

template<class ValType>

bool TMatrix<ValType>::operator==(const TMatrix<ValType>& mt) const

{ //тело функции представлено в приложении }

Для функции с возвращающим значением типа bool по сравнению на равенство двух матриц в качестве параметров приходит постоянная ссылка на объект.

**3.5.2 operator** !=

template<class ValType>

bool TMatrix<ValType>::operator!=(const TMatrix<ValType>& mt) const

{ //тело функции представлено в приложении }

Для функции с возвращающим значением типа bool по сравнению на неравенство двух матриц в качестве параметров приходит постоянная ссылка на объект.

**3.5.3 operator =**

template<typename ValType>

TMatrix<ValType>& TMatrix<ValType>::operator=(const TMatrix<ValType>& mt)

{ //тело функции представлено в приложении }

Для функции типа класса по присваиванию в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.5.4 operator +**

template<typename ValType>

TMatrix<ValType> TMatrix<ValType>::operator+(const TMatrix& mt)

{ return TVector<TVector<ValType> >::operator+(mt); }

Для функции типа класса по суммированию двух матриц в качестве параметров приходит постоянная ссылка на объект. Идет обращение к оператору суммированию векторов.

Для функции типа класса по оператору + = двух матриц в качестве параметров приходит постоянная ссылка на объект. Идет обращение к оператору + = векторов.

**3.5.5 operator –**

template<typename ValType>

TMatrix<ValType> TMatrix<ValType>::operator-(const TMatrix& mt)

{ return TVector<TVector<ValType> >::operator-(mt); }

Для функции типа класса по расчёту разности двух матриц в качестве параметров приходит постоянная ссылка на объект. Идет обращение к оператору вычитание векторов.

**3.5.6 istream**

friend std::istream& operator>>(std::istream& in, TMatrix& mt)

{

for (int i = 0; i < mt.Size; i++)

in >> mt.pVector[i];

return in;

}

Для функции ввода матрицы приходят 2 параметра.

**3.5.7 ostream**

friend std::ostream& operator<<(std::ostream& out, const TMatrix& mt)

{

for (int i = 0; i < mt.Size; i++)

out << mt.pVector[i] << std::endl;

return out;

}

Для функции вывода матрицы приходят 2 параметра.

# **Эксперименты**

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

Рассмотрим код отвечающий за суммирование матриц:

Matrix operator+(Matrix &mt)  
{  
 Matrix res(n,Size);  
 for(int i = 0; i<n; ++i)  
 for(int j = 0; j<Size; ++j)  
 res.pVector[i][j]=pVector[i][j]+mt.pVector[i][j];  
 return res;  
 }

Асимптотическая сложность этого кода:

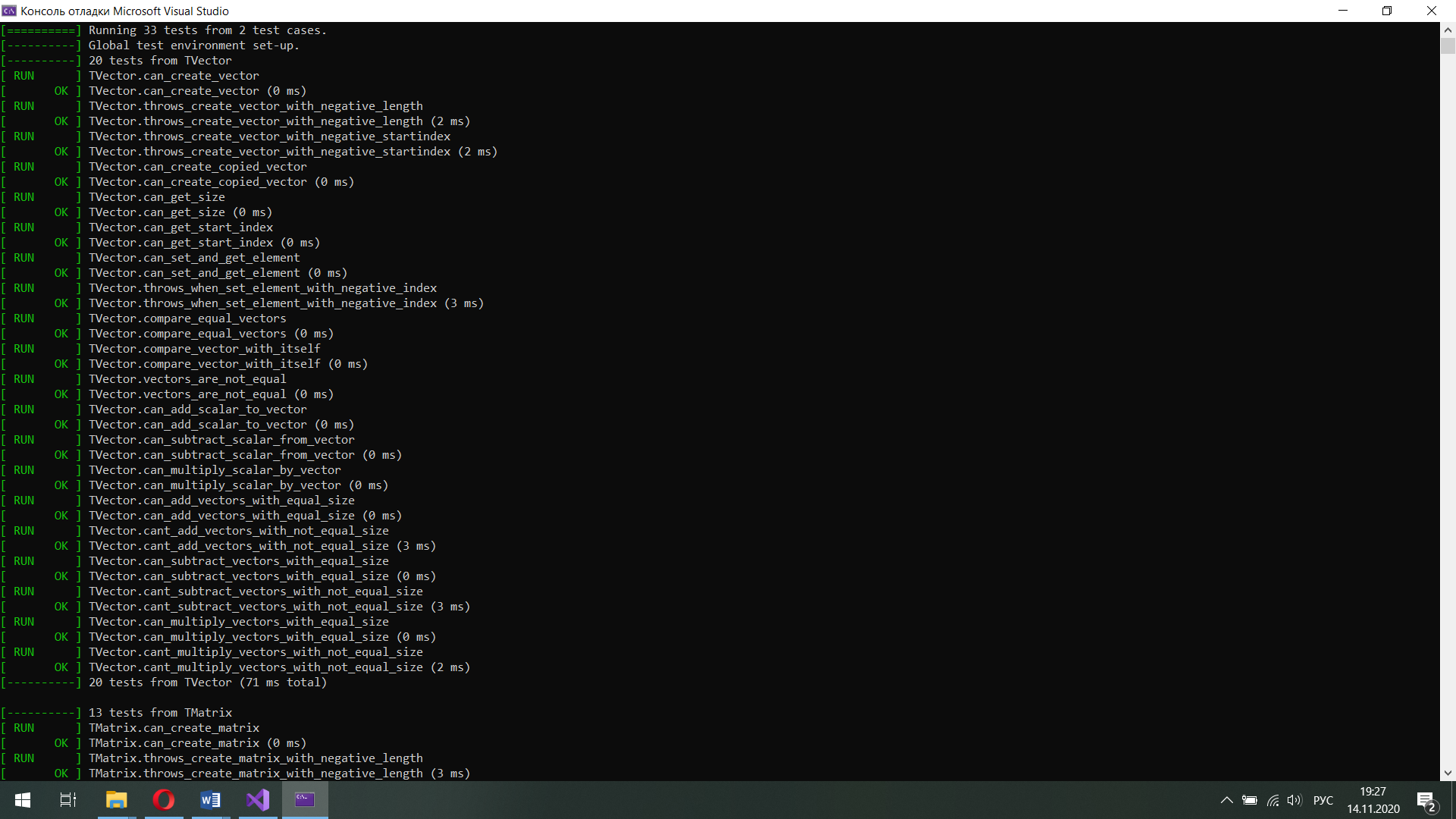
Теперь мы произведем замеры времени сложения матриц:

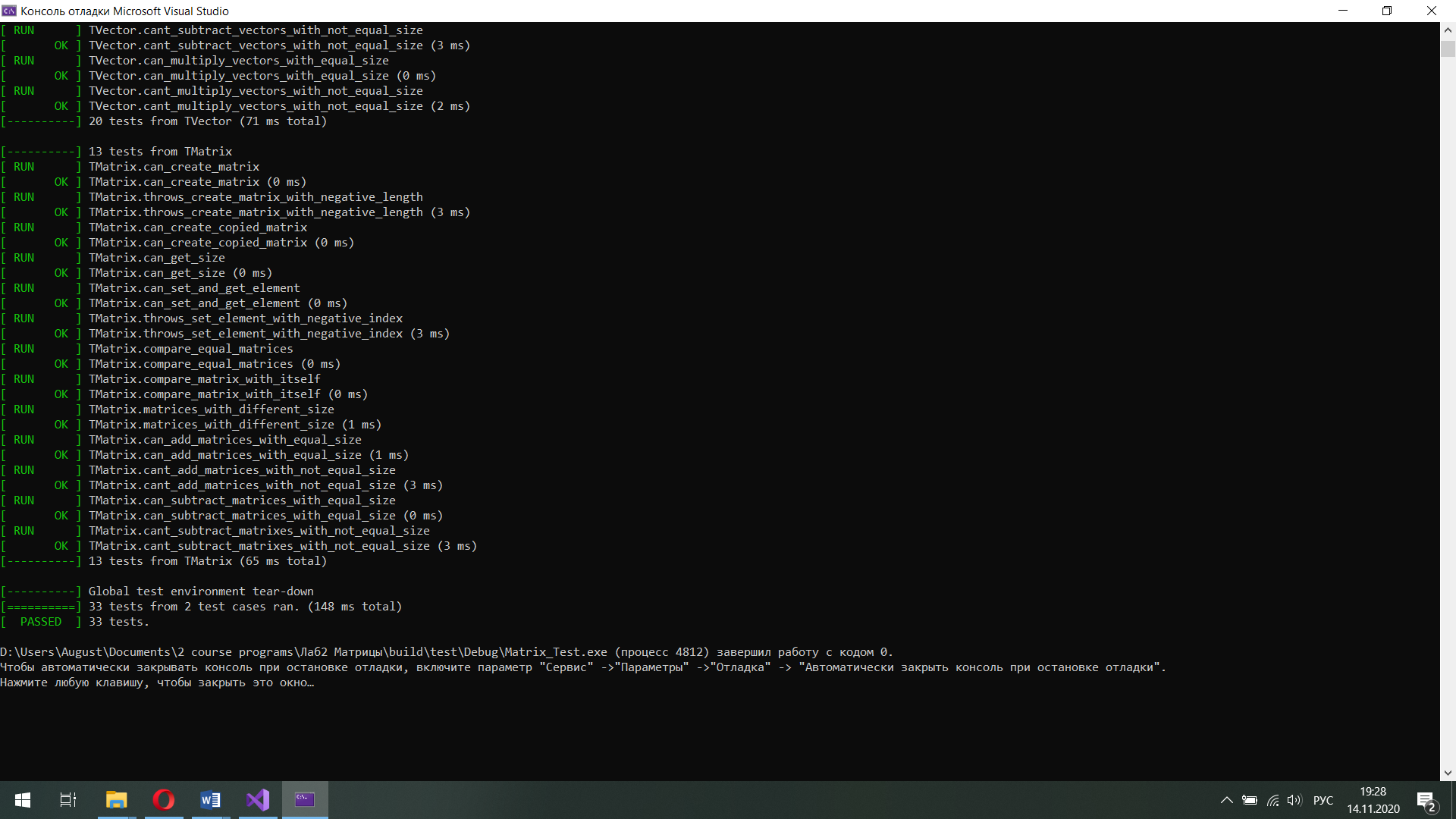
|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 0.73 |
| 2000 | 2.24 |
| 4000 | 9.03 |

То есть при увеличении количества элементов в 2 раза относительно предыдущего, время увеличивается примерно в 4 раза.

# **Заключение**

В модулях test\_main.cpp, test\_Matrix.cpp, test\_Vector.cpp прописаны тесты для соответственного модуля класса, которые позволяют проверить правильность выполнения программы. Сами тесты представлены в приложении. Результат их прохождения представлен ниже.





Написанная программа, поддерживающая эффективное хранение квадратных матриц и удовлетворяющая указанным допущениям, верно выполняет необходимые операции.

Целью работы была реализация структуры данных для хранения квадратных матриц. В результате была написана структура данных, в которой уменьшены затраты памяти для хранения векторов. Были проведены арифметические операции с матрицами, реализованных в данной структуре.

# **Список литературы**

1. Материалы по лабораторным работам

http://www.unn.ru/books/met\_files/Pract\_ADS.pdf

2. Рабочие материалы студента по общему курсу «Методы программирования»

http://www.itmm.unn.ru/files/2019/02/Metody-programmirovaniya-1-chast-2015.pdf

3. Лабораторный практикум

http://www.unn.ru/books/met\_files/Pract\_ADS.pdf

# **Приложение**

**TVector.h**

#ifndef \_TVECTOR\_H\_

#define \_TVECTOR\_H\_

#include <iostream>

template <class ValType>

class TVector

{

protected:

ValType\* pVector;

int Size; // размер вектора

int StartIndex; // индекс первого элемента вектора

public:

TVector(int s = 10, int si = 0);

TVector(const TVector& v); // конструктор копировани¤

~TVector();

int GetSize() { return Size; } // размер вектора

int GetStartIndex() { return StartIndex; } // индекс первого элемента

ValType& operator[](int pos); // доступ

bool operator==(const TVector& v) const; // сравнение

bool operator!=(const TVector& v) const; // сравнение

TVector& operator=(const TVector& v); // присваивание

// скалярные операции

TVector operator+(const ValType& val); // прибавить скаляр

TVector operator-(const ValType& val); // вычесть скаляр

TVector operator\*(const ValType& val); // умножить на скаляр

// векторные операции

TVector operator+(const TVector& v); // сложение

TVector operator-(const TVector& v); // вычитание

ValType operator\*(const TVector& v); // скалярное произведение

// ввод-вывод

friend std::istream& operator>>(std::istream& in, TVector& v)

{

for (int i = 0; i < v.Size; i++)

in >> v.pVector[i];

return in;

}

friend std::ostream& operator<<(std::ostream& out, const TVector& v)

{

for (int i = 0; i < v.Size; i++)

out << v.pVector[i] << ' ';

return out;

}

};

template<typename ValType>

TVector<ValType>::TVector(int s, int si)

: Size(s), StartIndex(si)

{

if (Size < 0)

throw "negative vector size";

if (StartIndex < 0)

throw "negative startindex";

pVector = new ValType[s];

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

pVector[i] = 0;

}

template<typename ValType>

TVector<ValType>::TVector(const TVector& v)

: Size(v.Size), StartIndex(v.StartIndex), pVector(nullptr)

{

pVector = new ValType[Size];

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

pVector[i] = v.pVector[i];

}

template<typename ValType>

TVector<ValType>::~TVector()

{

if (pVector != nullptr)

delete[] pVector;

}

template<typename ValType>

bool TVector<ValType>::operator==(const TVector& v) const

{

if ((Size != v.Size) || (StartIndex != v.StartIndex))

return false;

for (int i = StartIndex; i < Size; i++)

if (pVector[i] != v.pVector[i])

return false;

return true;

}

template<typename ValType>

bool TVector<ValType>::operator!=(const TVector& v) const

{

return !(\*this == v);

}

template<typename ValType>

TVector<ValType>& TVector<ValType>::operator=(const TVector& v)

{

if (this != &v)

{

if (Size != v.Size)

{

Size = v.Size;

delete[] pVector;

pVector = new ValType[Size];

}

StartIndex = v.StartIndex;

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

pVector[i] = v.pVector[i];

}

return \*this;

}

template<typename ValType>

TVector<ValType> TVector<ValType>::operator+(const ValType& val)

{

TVector result(Size, StartIndex);

for (int i = StartIndex; i < Size; i++)

result.pVector[i] = pVector[i] + val;

return result;

}

template<typename ValType>

TVector<ValType> TVector<ValType>::operator-(const ValType& val)

{

TVector result(Size, StartIndex);

for (int i = StartIndex; i < Size; i++)

result.pVector[i] = pVector[i] - val;

return result;

}

template<typename ValType>

TVector<ValType> TVector<ValType>::operator\*(const ValType& val)

{

TVector result(Size, StartIndex);

for (int i = StartIndex; i < Size; i++)

result.pVector[i] = pVector[i] \* val;

return result;

}

template<typename ValType>

TVector<ValType> TVector<ValType>::operator+(const TVector& v)

{

if ((Size != v.Size) || (StartIndex != v.StartIndex))

throw "Different sizes of vectors";

TVector result(Size, StartIndex);

for (int i = StartIndex; i < Size; i++)

result.pVector[i] = pVector[i] + v.pVector[i];

return result;

}

template<typename ValType>

TVector<ValType> TVector<ValType>::operator-(const TVector& v)

{

if ((Size != v.Size) || (StartIndex != v.StartIndex))

throw "Different sizes of vectors";

TVector result(Size, StartIndex);

for (int i = StartIndex; i < Size; i++)

result.pVector[i] = pVector[i] - v.pVector[i];

return result;

}

template<typename ValType>

ValType TVector<ValType>::operator\*(const TVector& v)

{

if ((Size != v.Size) || (StartIndex != v.StartIndex))

throw "Different sizes of vectors";

ValType result = 0;

for (int i = StartIndex; i < Size; i++)

result += pVector[i] \* v.pVector[i];

return result;

}

template<typename ValType>

ValType& TVector<ValType>::operator[](int pos)

{

if ((pos < StartIndex) || (pos >= Size))

throw "Wrong position";

return pVector[pos];

}

#endif

**TMatrix.h**

#ifndef \_TMATRIX\_H\_

#define \_TMATRIX\_H\_

#include "TVector.h"

template <class ValType>

class TMatrix : public TVector<TVector<ValType>>

{

using TVector<TVector<ValType>>::pVector;

using TVector<TVector<ValType>>::Size;

using TVector<TVector<ValType>>::StartIndex;

public:

TMatrix(int s = 10);

TMatrix(const TMatrix& mt); // копирование

TMatrix(const TVector<TVector<ValType>>& mt); // преобразование типа

bool operator==(const TMatrix& mt) const; // сравнение

bool operator!=(const TMatrix& mt) const; // сравнение

TMatrix& operator= (const TMatrix& mt); // присваивание

TMatrix operator+ (const TMatrix& mt); // сложение

TMatrix operator- (const TMatrix& mt); // вычитание

// ввод / вывод

friend std::istream& operator>>(std::istream& in, TMatrix& mt)

{

for (int i = 0; i < mt.Size; i++)

in >> mt.pVector[i];

return in;

}

friend std::ostream& operator<<(std::ostream& out, const TMatrix& mt)

{

for (int i = 0; i < mt.Size; i++)

out << mt.pVector[i] << std::endl;

return out;

}

};

template<typename ValType>

TMatrix<ValType>::TMatrix(int s) : TVector<TVector<ValType>>(s)

{

if (s > 0)

{

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

pVector[i] = TVector<ValType>(s - i, i);

}

else throw - 1;

}

template<typename ValType>

TMatrix<ValType>::TMatrix(const TMatrix<ValType>& mt)

: TVector<TVector<ValType>>(mt)

{

}

template<typename ValType>

TMatrix<ValType>::TMatrix(const TVector<TVector<ValType>>& mt)

: TVector<TVector<ValType>>(mt)

{

}

template<typename ValType>

bool TMatrix<ValType>::operator==(const TMatrix<ValType>& mt) const

{

if (Size != mt.Size)

return false;

for (int i = 0; i < mt.Size; i++)

if (pVector[i] != mt.pVector[i])

return false;

return true;

}

template<typename ValType>

bool TMatrix<ValType>::operator!=(const TMatrix<ValType>& mt) const

{

return !(\*this == mt);

}

template<typename ValType>

TMatrix<ValType>& TMatrix<ValType>::operator=(const TMatrix<ValType>& mt)

{

if (this != &mt)

{

if (Size != mt.Size)

{

delete[] pVector;

pVector = new TVector<ValType>[mt.Size];

}

Size = mt.Size;

StartIndex = mt.StartIndex;

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

pVector[i] = mt.pVector[i];

}

return \*this;

}

template<typename ValType>

TMatrix<ValType> TMatrix<ValType>::operator+(const TMatrix& mt)

{

return TVector<TVector<ValType> >::operator+(mt);

}

template<typename ValType>

TMatrix<ValType> TMatrix<ValType>::operator-(const TMatrix& mt)

{

return TVector<TVector<ValType> >::operator-(mt);

}

#endif

**Main.cpp**

#include <iostream>

#include "TMatrix.h"

int main()

{

TMatrix<int> a(5), b(5), c(5);

for (int i = 0; i < 5; i++)

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

{

a[i][j] = i \* 10 + j;

b[i][j] = (i \* 10 + j) \* 100;

}

c = a + b;

std::cout << "Matrix a = " << std::endl << a << std::endl;

std::cout << "Matrix b = " << std::endl << b << std::endl;

std::cout << "Matrix c = a + b" << std::endl << c << std::endl;

}

**Test\_main.cpp**

#include <gtest.h>

int main(int argc, char \*\*argv) {

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

**Test\_Vector.cpp**

#include "TVector.h"

#include <gtest.h>

TEST(TVector, can\_create\_vector)

{

ASSERT\_NO\_THROW(TVector<int> V(5));

}

TEST(TVector, throws\_create\_vector\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TVector<int> V(-5));

}

TEST(TVector, throws\_create\_vector\_with\_negative\_startindex)

{

ASSERT\_ANY\_THROW(TVector<int> V(5, -1));

}

TEST(TVector, can\_create\_copied\_vector)

{

TVector<int> V(5);

ASSERT\_NO\_THROW(TVector<int> V1(V));

}

TEST(TVector, can\_get\_size)

{

TVector<int> V(5);

EXPECT\_EQ(5, V.GetSize());

}

TEST(TVector, can\_get\_start\_index)

{

TVector<int> V(5, 1);

EXPECT\_EQ(1, V.GetStartIndex());

}

TEST(TVector, can\_set\_and\_get\_element)

{

TVector<int> V(5);

V[0] = 2;

EXPECT\_EQ(2, V[0]);

}

TEST(TVector, throws\_when\_set\_element\_with\_negative\_index)

{

TVector<int> V(5);

ASSERT\_ANY\_THROW(V[-1] = 1;);

}

TEST(TVector, compare\_equal\_vectors)

{

TVector<int> V(5), V1(5);

for (int i = 0; i < 5; i++)

V[i] = 1;

for (int i = 0; i < 5; i++)

V1[i] = 1;

EXPECT\_EQ(1, V1 == V);

}

TEST(TVector, compare\_vector\_with\_itself)

{

TVector<int> V(5);

for (int i = 0; i < 5; i++)

V[i] = 1;

EXPECT\_EQ(1, V == V);

}

TEST(TVector, vectors\_are\_not\_equal)

{

TVector<int> V(5), V1(3);

EXPECT\_NE(1, V == V1);

}

TEST(TVector, can\_add\_scalar\_to\_vector)

{

TVector<int> V(5);

for (int i = 0; i < 5; i++)

V[i] = 1;

TVector<int> SUM(5);

SUM = V + 2;

TVector<int> RES(5);

for (int i = 0; i < 5; i++)

RES[i] = 3;

EXPECT\_EQ(1, SUM == RES);

}

TEST(TVector, can\_subtract\_scalar\_from\_vector)

{

TVector<int> V(5);

for (int i = 0; i < 5; i++)

V[i] = 3;

TVector<int> SUB(5);

SUB = V - 2;

TVector<int> RES(5);

for (int i = 0; i < 5; i++)

RES[i] = 1;

EXPECT\_EQ(1, SUB == RES);

}

TEST(TVector, can\_multiply\_scalar\_by\_vector)

{

TVector<int> V(5);

for (int i = 0; i < 5; i++)

V[i] = 3;

TVector<int> MUL(5);

MUL = V \* 2;

TVector<int> RES(5);

for (int i = 0; i < 5; i++)

RES[i] = 6;

EXPECT\_EQ(1, MUL == RES);

}

TEST(TVector, can\_add\_vectors\_with\_equal\_size)

{

TVector<int> V1(3), V2(3);

V1[0] = 1; V1[1] = 2; V1[2] = 3;

V2[0] = 2; V2[1] = 3; V2[2] = 4;

TVector<int> SUM(3);

SUM = V1 + V2;

TVector<int> RES(3);

RES[0] = 3; RES[1] = 5; RES[2] = 7;

EXPECT\_EQ(1, SUM == RES);

}

TEST(TVector, cant\_add\_vectors\_with\_not\_equal\_size)

{

TVector<int> V1(5), V2(3);

ASSERT\_ANY\_THROW(V1 + V2);

}

TEST(TVector, can\_subtract\_vectors\_with\_equal\_size)

{

TVector<int> V1(3), V2(3);

V1[0] = 1; V1[1] = 2; V1[2] = 3;

V2[0] = 2; V2[1] = 3; V2[2] = 4;

TVector<int> SUB(3);

SUB = V2 - V1;

TVector<int> RES(3);

RES[0] = 1; RES[1] = 1; RES[2] = 1;

EXPECT\_EQ(1, SUB == RES);

}

TEST(TVector, cant\_subtract\_vectors\_with\_not\_equal\_size)

{

TVector<int> V1(5), V2(3);

ASSERT\_ANY\_THROW(V1 - V2);

}

TEST(TVector, can\_multiply\_vectors\_with\_equal\_size)

{

TVector<int> V1(3), V2(3);

V1[0] = 1; V1[1] = 2; V1[2] = 3;

V2[0] = 2; V2[1] = 3; V2[2] = 4;

int MUL;

MUL = V1 \* V2;

int RES = 20;

EXPECT\_EQ(1, MUL == RES);

}

TEST(TVector, cant\_multiply\_vectors\_with\_not\_equal\_size)

{

TVector<int> V1(5), V2(3);

ASSERT\_ANY\_THROW(V1 \* V2);

}

**Test\_Matrix.cpp**

#include "TMatrix.h"

#include "TVector.h"

#include <gtest.h>

TEST(TMatrix, can\_create\_matrix)

{

ASSERT\_NO\_THROW(TMatrix<int> M(5));

}

TEST(TMatrix, throws\_create\_matrix\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TMatrix<int> M(-5));

}

TEST(TMatrix, can\_create\_copied\_matrix)

{

TMatrix<int> M(5);

ASSERT\_NO\_THROW(TMatrix<int> M1(M));

}

TEST(TMatrix, can\_get\_size)

{

TMatrix<int> M(5);

ASSERT\_NO\_THROW(M.GetSize());

}

TEST(TMatrix, can\_set\_and\_get\_element)

{

TMatrix<int> M(5);

M[1][1] = 1;

EXPECT\_EQ(1, M[1][1]);

}

TEST(TMatrix, throws\_set\_element\_with\_negative\_index)

{

TMatrix<int> M(5);

ASSERT\_ANY\_THROW(M[1][-1] = 1);

}

TEST(TMatrix, compare\_equal\_matrices)

{

TMatrix<int> M1(5), M2(5);

EXPECT\_EQ(1, M1 == M2);

}

TEST(TMatrix, compare\_matrix\_with\_itself)

{

TMatrix<int> M(5);

EXPECT\_EQ(1, M == M);

}

TEST(TMatrix, matrices\_with\_different\_size)

{

TMatrix<int> M1(5), M2(3);

EXPECT\_NE(1, M1 == M2);

}

TEST(TMatrix, can\_add\_matrices\_with\_equal\_size)

{

TMatrix<int> M1(2), M2(2);

TMatrix<int> SUM(2);

M1[0][0] = 1;

M2[0][0] = 2;

SUM = M1 + M2;

TMatrix<int> RES(2);

RES[0][0] = 3;

EXPECT\_EQ(1, SUM == RES);

}

TEST(TMatrix, cant\_add\_matrices\_with\_not\_equal\_size)

{

TMatrix<int> M1(5), M2(3);

ASSERT\_ANY\_THROW(M1 + M2);

}

TEST(TMatrix, can\_subtract\_matrices\_with\_equal\_size)

{

TMatrix<int> M1(2), M2(2);

TMatrix<int> SUB(2);

M1[0][0] = 1;

M2[0][0] = 2;

SUB = M1 - M2;

TMatrix<int> RES(2);

RES[0][0] = -1;

EXPECT\_EQ(1, SUB == RES);

}

TEST(TMatrix, cant\_subtract\_matrixes\_with\_not\_equal\_size)

{

TMatrix<int> M1(5), M2(3);

ASSERT\_ANY\_THROW(M1 - M2);

}