МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

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**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Структуры хранения для матриц специального вида»**

**Выполнила:** студентка группы 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Ясакова Т.Е.

Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д./

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc153830579)

[1 Постановка задачи 4](#_Toc153830580)

[2 Руководство пользователя 5](#_Toc153830581)

[2.1 Приложение для демонстрации работы вектора 5](#_Toc153830582)

[2.2 Приложение для демонстрации работы матрицы 5](#_Toc153830583)

[3 Руководство программиста 7](#_Toc153830584)

[3.1 Описание алгоритмов 7](#_Toc153830585)

[3.1.1 Вектор 7](#_Toc153830586)

[3.1.2 Матрица 9](#_Toc153830587)

[3.2 Описание программной реализации 11](#_Toc153830588)

[3.2.1 Описание класса TVector 11](#_Toc153830589)

[3.2.2 Описание класса TMatrix 15](#_Toc153830590)

[Заключение 18](#_Toc153830591)

[Литература 19](#_Toc153830592)

[Приложения 20](#_Toc153830593)

[Приложение А. Реализация класса TVector 20](#_Toc153830594)

[Приложение Б. Реализация класса TMatrix 23](#_Toc153830595)

# Введение

Лабораторная работа "Векторы и верхнетреугольные матрицы на шаблонах" направлена на изучение и практическое применение концепции шаблонов в языке программирования C++. Шаблоны позволяют создавать обобщенные типы данных, которые могут быть использованы с различными типами данных без необходимости дублирования кода. В рамках данной работы мы будем рассматривать примеры использования шаблонов для реализации векторов и верхнетреугольных матриц.

# Постановка задачи

**Цель:**

Ознакомление студентов с принципами работы шаблонов в языке C++ и их применением для создания обобщенных типов данных. В результате выполнения работы студенты должны приобрести практические навыки по созданию и использованию шаблонов для реализации векторов и верхнетреугольных матриц.

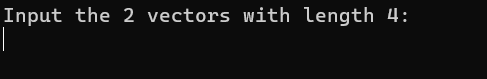
**Задачи:**

1. Изучение основных принципов работы шаблонов в языке C++.
2. Разработка шаблонного класса для реализации вектора, который будет поддерживать основные операции.
3. Разработка шаблонного класса для реализации верхнетреугольной матрицы, который будет поддерживать операции сложения матриц, умножения матриц и т.д.
4. Проведение тестирования разработанных шаблонных классов на различных наборах данных для проверки их корректности и эффективности.

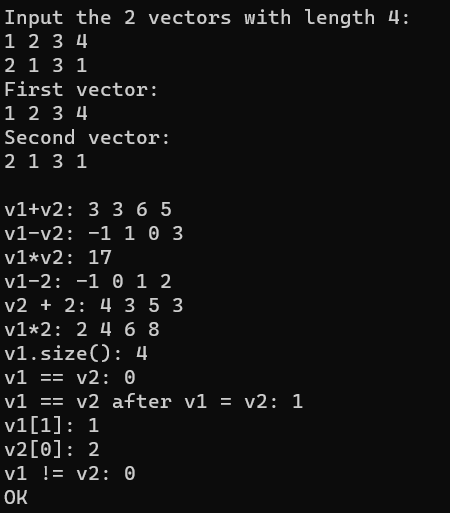
# Руководство пользователя

## Приложение для демонстрации работы вектора

1. Запустите приложение с названием sample\_tvector.exe. В результате появится окно, показанное ниже и вам будет предложено ввести 2 целочисленных вектора длины 4 (рис. 1).



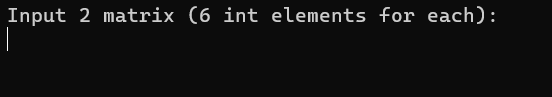
1. Основное окно программы
2. После ввода будет выведены результаты соответствующих операций и функций (рис. 2).



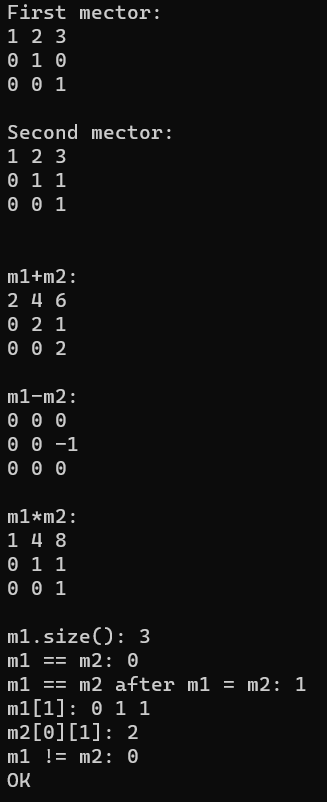
1. Результат тестирования функций класса TVector

## Приложение для демонстрации работы матрицы

1. Запустите приложение с названием sample\_tmatrix.exe. В результате появится окно, показанное ниже, вам будет предложено ввести 2 целочисленных верхнетреугольных матрицы 3 х 3 (вам нужно ввести 12 чисел) (рис. 3).



1. Основное окно программы
2. После ввода матриц будет выведены результаты соответствующих операций и функций (рис. 4).



1. Результат тестирования функций класса TMatrix

# Руководство программиста

## Описание алгоритмов

### Вектор

Вектор – структура хранения. Он хранит элементы одного типа данных.

Вектор хранится в виде массива элементов одного типа данных, стартового индекса и количества элементов в векторе. Такая структура позволяет эффективно работать с матричными операциями.

Если стартовый индекс отличен от нуля, то все элементы от 0 до стартового индекса (не включительно) будут равны нейтральному элементу (нулю).

Пример целочисленного вектора: стартового индекса 1, и размера 4: (0, 1, 2, 3, 4).

Вектор поддерживает операции сложения, вычитания и умножения с элементом типа данных, сложения, вычитания, скалярного произведения с вектором того же типа данных, операции индексации, сравнение на равенство (неравенство).

**Операция сложения**

Операция сложения определена для вектора того же типа (складываются элементы первого и второго вектора с одинаковыми индексами) или некоторого элемента того же типа (каждый элемент вектора отдельно складывается с элементом).

Пример:

V = {1, 2, 3, 4, 5}

Сложение с вектором:

V1 = {1, 1, 1, 1, 1}

V + V1 = {2, 3, 4, 5, 6}

Сложение с константой:

c = 2

V + c = {3, 4, 5, 6, 7}

**Операция вычитания**

Операция вычитания определена для вектора того же типа (вычитаются элементы первого и второго вектора с одинаковыми индексами) или некоторого элемента того же типа (каждый элемент вектора отдельно вычитается с элементом).

Пример:

V = {1, 2, 3, 4, 5}

Вычитание с вектором:

V1 = {1, 1, 1, 1, 1}

V - V1 == {0, 1, 2, 3, 4}

Вычитание с константой:

c = 2

V - c = {-1, 0, 1, 2, 3}

**Операция умножения**

Операция умножения определена для вектора того же типа (скалярное произведение векторов) или некоторого элемента того же типа (каждый элемент вектора отдельно умножается с элементом).

Пример:

V = {1, 2, 3, 4, 5}

Сложение с вектором:

V1 = {1, 1, 1, 1, 1}

V \* V1 = 1\*1 + 2\*1 + 3\*1 + 4\*1 + 5\*1 = 15

Сложение с константой:

c = 2

V \* c = {2, 4, 6, 8, 10}

**Операция индексации**

Операция индексации предназначена для получения элемента вектора. Причем, если позиция будет меньше, чем стартовый индекс, то будет выведено исключение.

Пример:

V = {0, 1, 2, 3, 4}

Получение индекса 1:

V[1] = 1

**Операция сравнения на равенство**

Операция сравнения на равенство с вектором возвращает 1, если вектора равны поэлементно, причём их стартовые индексы и размеры тоже равны, 0 в противном случае.

Пример

V = {1, 2, 3, 4, 5}, V1 = {1, 2, 3, 4, 5}, V2 = {0, 1, 2, 3, 4}

Сложение с вектором:

(V == V1) = 1

(V == V2) = 0

**Операция сравнения на неравенство**

Операция сравнения на равенство с вектором возвращает 0, если вектора равны поэлементно, причём их стартовые индексы и размеры тоже равны, 1 в противном случае.

Пример

V = {1, 2, 3, 4, 5}, V1 = {1, 2, 3, 4, 5}, V2 = {0, 1, 2, 3, 4}

Сложение с вектором:

(V == V1) = 0

(V == V2) = 1

### Матрица

Матрица – вектор векторов, структура хранения. Она хранит элементы одного типа данных.

Матрица хранится в виде массива векторов, стартового индекса и количества элементов в матрице (именно количество столбцов или строк, т.к. матрица квадратная и верхнетреугольная.

Пример целочисленной матрицы 3х3:

1 1 1

0 2 2

0 0 3

Матрица поддерживает операции сложения, вычитания и умножения с матрицей того же типа данных, операции индексации, сравнение на равенство (неравенство).

**Операция сложения**

Операция сложения определена для матрицы того же типа (складываются элементы первой и второй матрицы с одинаковыми индексами).

Пример:

**Операция вычитания**

Операция вычитания определена для матрицы того же типа (вычитаются элементы первой и второй матрицы с одинаковыми индексами).

Пример:

**Операция умножения**

Операция умножения определена для матрицы того же типа (скалярное произведение векторов).

**Операция индексации**

Операция индексации предназначена для получения элемента матрицы. Причем, Элемент матрицы – вектор-строка, также можно вывести элемент матрицы по индексу, т.к. для вектора также перегружена операция индексации.

M[0] = {1, 2, 3}

M[0][1] = 2

**Операция сравнения на равенство**

Операция сравнения на равенство с матрицей возвращает 1, если они равны поэлементно, причём их стартовые индексы и размеры тоже равны, 0 в противном случае.

Пример:

**Операция сравнения на равенство**

Операция сравнения на равенство с матрицей возвращает 0, если они равны поэлементно, причём их стартовые индексы и размеры тоже равны, 1 в противном случае.

Пример:

## Описание программной реализации

### Описание класса TVector

template <class ValType>

class TVector

{

private:

int StartIndex;

protected:

ValType \*pVector;

int Size;

public:

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

TVector(const TVector& v);

virtual ~TVector();

int GetSize() { return Size; }

int GetStartIndex() { return StartIndex; }

ValType& operator[](int pos);

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

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

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

// скалярные операции

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

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

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

// векторные операции

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

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

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

// ввод-вывод

friend istream& operator>>(istream &in, TVector<ValType> &v);

friend ostream& operator<<(ostream &out,TVector<ValType> &v);

};

Назначение: представление вектора

Поля:

StartIndex – индекс первого необходимого элемента вектора.

\*pVector – память для представления элементов вектора.

Size – количество нужных элементов вектора.

Методы:

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

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры:

s – длинна вектора (по умолчанию 10).

si – стартовый индекс (по умолчанию 0).

Выходные параметры: отсутствуют.

TVector(const TVector<ValType>& v);

Назначение: конструктор копирования.

Входные параметры:

v – экземпляр класса, на основе которого создаем новый объект.

Выходные параметры: отсутствуют.

virtual ~TVector();

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

int GetSize();

Назначение: получение размера вектора.

Входные параметры: отсутствуют.

Выходные параметры: размер вектора ( количество элементов).

int GetStartIndex();

Назначение: получение стартового индекса.

Входные параметры: отсутствуют.

Выходные параметры: стартовый индекс.

ValType& operator[](int pos);

Назначение: перегрузка оператора индексации.

Входные параметры:

pos – позиция (индекс) элемента.

Выходные параметры: элемент, который находится на pos позиции.

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

Назначение: оператор сравнения.

Входные параметры:

v – экземпляр класса, с которым сравниваем.

Выходные параметры:

true (1), если они равны, иначе false(0).

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

Назначение: оператор сравнения.

Входные параметры:

v – экземпляр класса, с которым сравниваем.

Выходные параметры:

true (1), если они не равны, иначе false(0).

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

Назначение: оператор присваивания.

Входные параметры:

v – экземпляр класса, который присваиваем.

Выходные параметры:

Ссылка на присвоенный экземпляр класса.

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

Назначение: оператор суммирования вектора и значения.

Входные параметры:

Val – элемент, с которым суммируем.

Выходные параметры:

Экземпляр класса, элементы которого на val больше.

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

Назначение: оператор вычитания вектора и значения.

Входные параметры:

Val – элемент, который вычитаем.

Выходные параметры:

Экземпляр класса, элементы которого на val меньше.

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

Назначение: оператор умножения вектора на значение.

Входные параметры:

Val – элемент, на который умножаем вектор.

Выходные параметры:

Экземпляр класса, элементы которого в val раз больше.

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

Назначение: оператор суммирования векторов.

Входные параметры:

V – вектор, который суммируем.

Выходные параметры:

Экземпляр класса, равный сумме двух векторов.

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

Назначение: оператор вычитания векторов.

Входные параметры:

V – вектор, который вычитаем.

Выходные параметры:

Экземпляр класса, равный разности двух векторов.

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

Назначение: оператор умножения векторов.

Входные параметры:

V – вектор, на который умножаем.

Выходные параметры:

Значение, равное скалярному произведению двух векторов.

friend istream& operator>>(istream &in, TVector<ValType> &v);

Назначение: оператор ввода вектора.

Входные параметры:

in – ссылка на буфер, из которого вводим вектор.

V – ссылка на вектор, который вводим.

Выходные данные:

in – ссылка буфер.

friend ostream& operator<<(ostream &out,TVector<ValType> &v);

Назначение: оператор вывода вектора

Входные параметры:

in – ссылка на буфер, из которого выводим вектор.

v – ссылка на вектор, который выводим.

Выходные данные:

in – ссылка буфер.

### Описание класса TMatrix

template <class ValType>

class TMatrix : public TVector<TVector<ValType>>

{

public:

TMatrix(int s = 10);

TMatrix(const TMatrix &mt);

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

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

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

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

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

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

TMatrix operator\*(const TMatrix<ValType> &mt);

// ввод / вывод

friend istream& operator>>(istream &in, TMatrix<ValType>&mt);

friend ostream & operator<<( ostream &out, const);

};

Класс наследуется от класса TVector<TVector<ValType>> (Public наследование).

Назначение: представление матрицы как вектор векторов

Поля:

StartIndex – индекс первого необходимого элемента.

\*pVector – память для представления элементов матрицы.

Size – размерность матрицы.

Методы:

TMatrix(int s = 10);

Назначение: конструктор по умолчанию и конструктор с параметрами

Входные параметры:

s – длина вектора (по умолчанию 10).

Выходные параметры: отсутствуют.

TMatrix(const TMatrix &mt);

Назначение: конструктор копирования

Входные параметры:

mt – экземпляр класса, на основе которого создаем новый объект.

Выходные параметры: отсутствуют.

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

Назначение: Конструктор преобразования типов.

Входные параметры:

mt – ссылка на TVector<TVector<ValType>> - на объект, который преобразуем.

Выходные данные: отсутствуют.

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

Назначение: оператор сравнения.

Входные параметры:

mt – экземпляр класса, с которым сравниваем.

Выходные параметры:

true (1), если они равны, иначе false(0).

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

Назначение: оператор сравнения.

Входные параметры:

mt – экземпляр класса, с которым сравниваем.

Выходные параметры:

true (1), если они не равны, иначе false(0).

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

Назначение: оператор присваивания.

Входные параметры:

mt – экземпляр класса, который присваиваем.

Выходные параметры:

Ссылка на присвоенный экземпляр класса.

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

Назначение: оператор суммирования матриц.

Входные параметры:

mt – ссылка на матрицу, которую суммируем.

Выходные параметры:

Экземпляр класса, равный сумме двух матриц.

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

Назначение: оператор вычитания матриц.

Входные параметры:

mt – ссылка на матрицу, которую вычитаем.

Выходные параметры:

Экземпляр класса, равный разности двух матриц.

TMatrix operator\*(const TMatrix<ValType> &mt);

Назначение: оператор умножения матриц.

Входные параметры:

mt – ссылка на матрицу, которую умножаем.

Выходные параметры:

Экземпляр класса, равный произведению двух матриц.

friend istream& operator>>(istream &in, TMatrix<ValType>&mt);

Назначение: оператор ввода матрицы.

Входные параметры:

in – ссылка на буфер, из которого вводим матрицу.

v – ссылка на матрицу, которую вводим.

Выходные данные:

in – ссылка буфер.

friend ostream& operator<<(ostream &out, TMatrix<ValType>&mt);

Назначение: оператор вывода матрицы.

Входные параметры:

out – ссылка на буфер, из которого выводим матрицу.

v – ссылка на матрицу, который выводим.

Выходные данные:

our – ссылка буфер.

# Заключение

В ходе выполнения лабораторной работы мы изучили и практически применили концепцию шаблонов в языке программирования C++. Шаблоны позволяют создавать обобщенные типы данных, которые могут быть использованы с различными типами данных без необходимости дублирования кода.

В рамках работы мы разработали шаблонный класс для реализации вектора, который поддерживает основные операции, такие как добавление элемента, удаление элемента, доступ к элементу по индексу и другие. Также мы разработали шаблонный класс для реализации верхнетреугольной матрицы, который поддерживает операции сложения матриц, умножения матрицы на матрицу и другие

# Литература

1. Треугольная матрица [https://ru.wikipedia.org/wiki/Треугольная\_матрица].
2. Лекция «Вектора и матрицы» Сысоев А.В [https://cloud.unn.ru/s/FkYBW5rJLDCgBmJ].

# Приложения

## Приложение А. Реализация класса TVector

#ifndef \_\_TVECTOR\_H\_\_

#define \_\_TVECTOR\_H\_\_

#include <iostream>

using namespace std;

const int MAX\_VECTOR\_SIZE = 100000;

// Шаблон вектора

template <class ValType>

class TVector

{

protected:

int StartIndex;

ValType \*pVector;

int Size;

public:

TVector<ValType>(int s = 10, int si = 0);

TVector<ValType>(const TVector& v);

virtual ~TVector<ValType>();

int GetSize() const { return Size; }

int GetStartIndex() const { return StartIndex; }

ValType& operator[](int pos);

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

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

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

// скалярные операции

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

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

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

// векторные операции

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

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

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

// ввод-вывод

friend istream& operator>>(istream &in, TVector<ValType> &v)

{

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

in >> v[i];

return in;

}

friend ostream& operator<<(ostream &out,TVector<ValType> &v)

{

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

{

out << ValType() << " ";

}

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

out << v[i] << " ";

}

return out;

}

};

template <class ValType>

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

{

if (s <= 0 || s > MAX\_VECTOR\_SIZE)

throw "Incorrect size";

if (si < 0)

throw "You cannot start at negative index!";

pVector = new ValType[s]();

}

template <class ValType>

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

{

Size = v.Size;

StartIndex = v.StartIndex;

pVector = new ValType[Size];

std::copy(v.pVector, v.pVector + v.Size, pVector);

}

template <class ValType>

TVector<ValType>::~TVector<ValType>()

{

delete[] pVector;

}

template <class ValType>

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

{

if (pos < 0 || pos>=MAX\_VECTOR\_SIZE)

throw "Wrong position";

if (pos < StartIndex)

throw "Wrong position (less than start index)";

if (pos - StartIndex < Size)

return pVector[pos - StartIndex];

throw "Access Error";

}

template <class ValType>

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

{

if ((StartIndex != v.StartIndex) || (Size != v.Size)) return false;

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

{

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

return false;

}

}

return true;

}

template <class ValType>

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

{

return !(\*this == v);

}

template <class ValType>

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

{

if (this == &v)

return \*this;

if (Size != v.Size)

{

delete[] pVector;

pVector = new ValType[v.Size];

}

Size = v.Size;

StartIndex = v.StartIndex;

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

{

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

}

return \*this;

}

template <class ValType>

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

{

TVector<ValType> A(Size, StartIndex);

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

{

A[i] = pVector[i] + val;

}

return A;

}

template <class ValType>

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

{

TVector<ValType> A(Size, StartIndex);

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

{

A[i] = pVector[i] - val;

}

return A;

}

template <class ValType>

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

{

TVector<ValType> A(Size , StartIndex);

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

{

A[i] = pVector[i] \* val;

}

return A;

}

template <class ValType>

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

{

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

throw "Size and StartIndex should be equal";

TVector<ValType> B(Size, StartIndex), tmp(v);

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

{

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

}

return B;

}

template <class ValType>

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

{

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

throw "Size and StartIndex should be equal";

TVector<ValType> B(Size, StartIndex), tmp(v);

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

{

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

}

return B;

}

template <class ValType>

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

{

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

throw "dimentions of vectors should be equal for dot product";

ValType ans=ValType();

TVector<ValType> tmp(v);

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

{

ans = ans + pVector[i] \* tmp[i];

}

return ans;

}

#endif

## Приложение Б. Реализация класса TMatrix

#ifndef \_\_TMATRIX\_H\_\_

#define \_\_TMATRIX\_H\_\_

#include <iostream>

#include "tvector.h"

const int MAX\_MATRIX\_SIZE = 100000;

//Наследуем матрицу от конкретного экземпляра TVector<ValType1>, где ValType1 = TVector<ValType>, причём поля в родительском классе.

template <class ValType>

class TMatrix : public TVector<TVector<ValType>>

{

public:

TMatrix<ValType>(int s = 10);

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

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

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

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

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

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

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

TMatrix<ValType> operator\*(const TMatrix<ValType> &mt);

//Эти операции изменят вид матрицы, что не логично для задачи.

//TMatrix& operator=(const ValType& v);

//TMatrix operator+(const ValType& v);

//TMatrix operator-(const ValType &v);

// ввод / вывод

friend istream& operator>>(istream &in, TMatrix<ValType>&mt)

{

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

in >> mt.pVector[i];

return in;

}

friend ostream & operator<<( ostream &out, const TMatrix<ValType>&mt)

{

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

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

return out;

}

};

template <class ValType>

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

{

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

{

TVector<ValType> x(Size - i, i);

pVector[i] = x;

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

}

}

template <class ValType>

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

TVector<TVector<ValType>>(mt)

{

}

template <class ValType>

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

TVector<TVector<ValType>>(mt)

{

}

template <class ValType>

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

{

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

}

template <class ValType>

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

{

return !(\*this == mt);

}

template <class ValType>

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

{

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

}

template <class ValType>

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

{

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

}

template <class ValType>

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

{

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

}

template <class ValType>

TMatrix<ValType> TMatrix<ValType>::operator\*(const TMatrix<ValType>& mt)

{

if (Size != mt.Size)

throw "Sizes should be equal!\n";

TMatrix<ValType> tmp(mt), res(Size);

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

{

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

{

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

{

res[i][j] += (\*this)[i][k] \* tmp[k][j];

}

}

}

return res;

}

#endif