

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
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»**

**Выполнил:** студент группы 3822Б1ФИ2  
\_\_\_\_\_ / Табунов И.Д.

Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП  
\_\_\_\_\_ / Кустикова В.Д./

Подпись

Нижний Новгород  
2023

# Содержание

Введение.....	3
1 Постановка задачи.....	4
2 Руководство пользователя.....	5
2.1 Приложение для демонстрации работы векторов .....	5
2.2 Приложение для демонстрации работы матриц .....	7
3 Руководство программиста .....	9
3.1 Описание алгоритмов .....	9
3.1.1 Вектор.....	9
3.1.2 Матрица.....	10
3.2 Описание программной реализации .....	12
3.2.1 Описание класса TVector.....	12
3.2.2 Описание класса TMatrix.....	15
Заключение .....	18
Литература .....	19
Приложения .....	20
Приложение А. Реализация класса TVector .....	20
Приложение Б. Реализация класса TMatrix .....	24
Приложение В. Sample_tmatrix.....	26
Приложение Г. Sample_tvector.....	26

## **Введение**

Данная лабораторная работа - изучение треугольных матриц и их представление в виде вектора, состоящего из векторов. Для реализации программы необходимо создать классы с шаблонами, различными функциями и перегрузить арифметические операторы. Треугольная матрица - это квадратная матрица, у которой все элементы ниже (или выше) главной диагонали равны нулю. Они используются для более компактного хранения данных.

# 1 Постановка задачи

## Цель:

реализовать классы для представления вектора `TVector` и верхнетреугольной матрицы `TMatrix` как вектора векторов.

## 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.

## 2 Руководство пользователя

### 2.1 Приложение для демонстрации работы векторов

1. Запустите приложение с названием `sample_tvector.exe`. В результате появится окно, показанное ниже, и вам будет предложено ввести длину векторов (рис. 1).

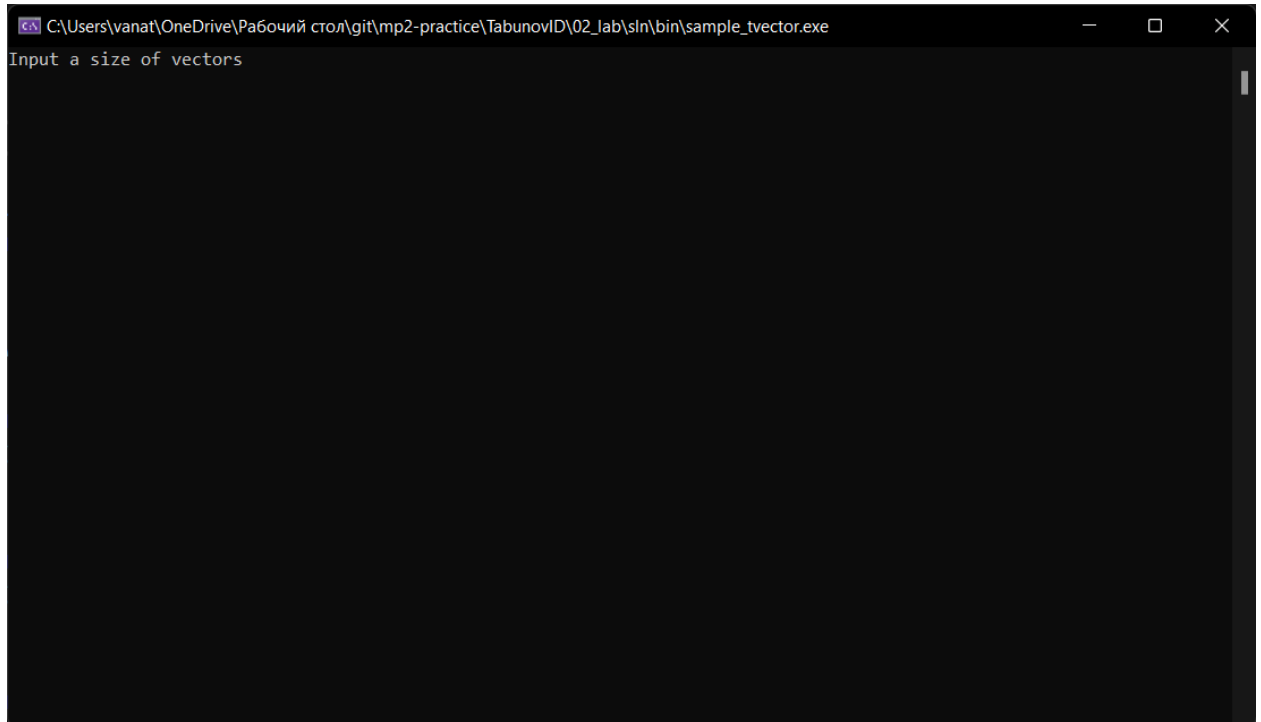


Рис. 1. Основное окно программы

2. После можно ввести два вектора длины, которую вы указали ранее (рис. 2).

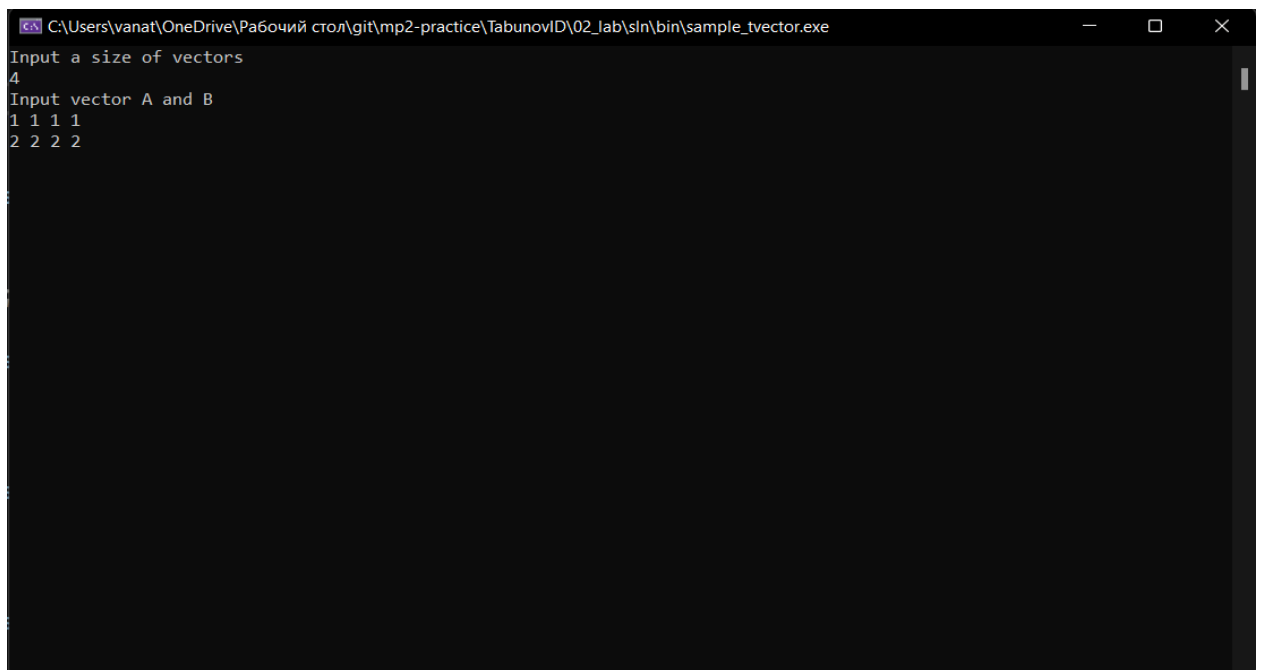
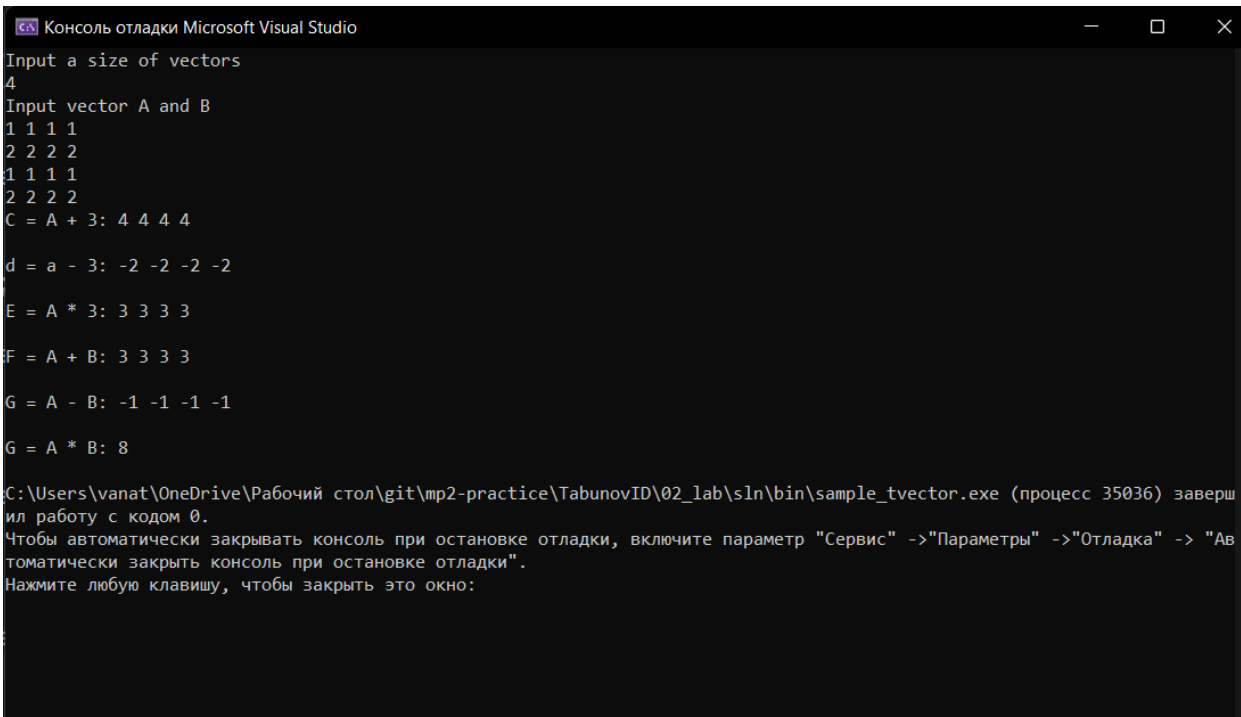


Рис. 2. Создание векторов

3. После ввода будут выведены результаты соответствующих операций и функций (рис. 3).



```
Консоль отладки Microsoft Visual Studio
Input a size of vectors
4
Input vector A and B
1 1 1 1
2 2 2 2
1 1 1 1
2 2 2 2
C = A + 3: 4 4 4 4
d = a - 3: -2 -2 -2 -2
E = A * 3: 3 3 3 3
F = A + B: 3 3 3 3
G = A - B: -1 -1 -1 -1
G = A * B: 8
C:\Users\vanat\OneDrive\Рабочий стол\git\mp2-practice\TabunovID\02_lab\sln\bin\sample_tvector.exe (процесс 35036) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Рис. 3. Получение размера векторов

## 2.2 Приложение для демонстрации работы матриц

1. Запустите приложение с названием `sample_tmatrix.exe`. В результате появится окно, показанное ниже и вам будет предложено ввести размерность матриц (рис. 44).

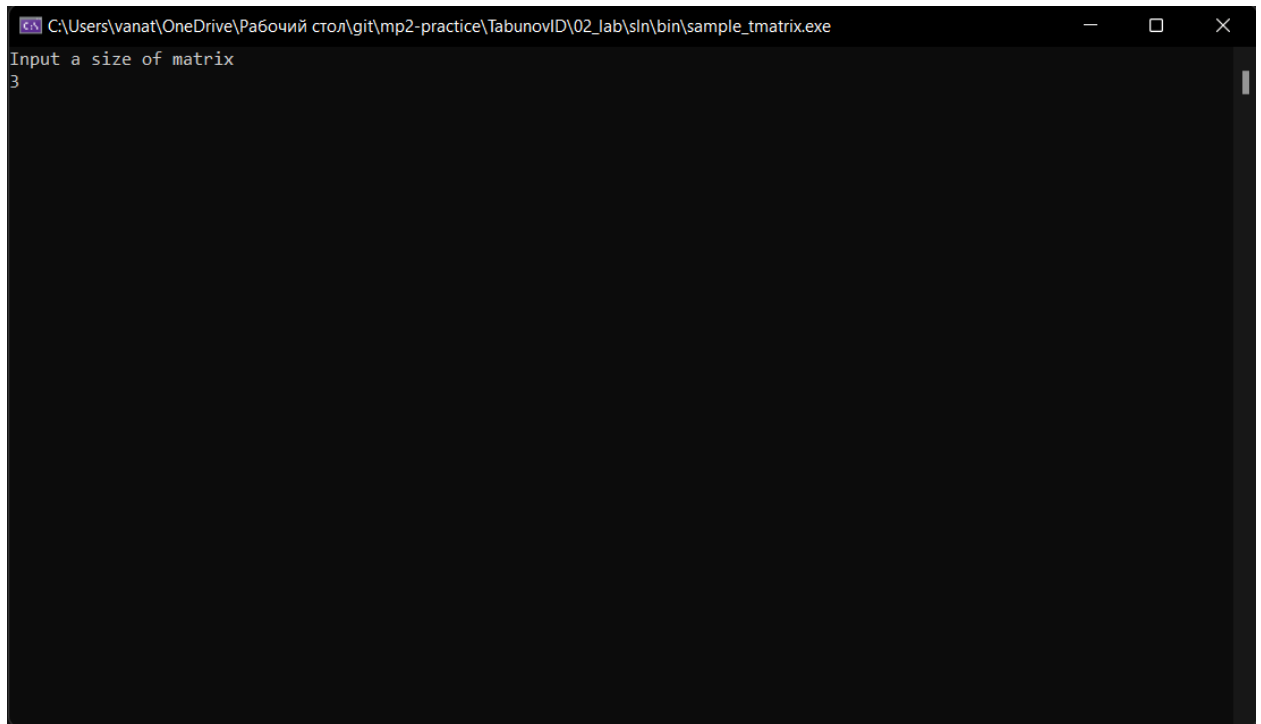


Рис. 4. Основное окно программы

2. вам будет предложено ввести 2 целочисленных верхнетреугольных матрицы (рис. 55).

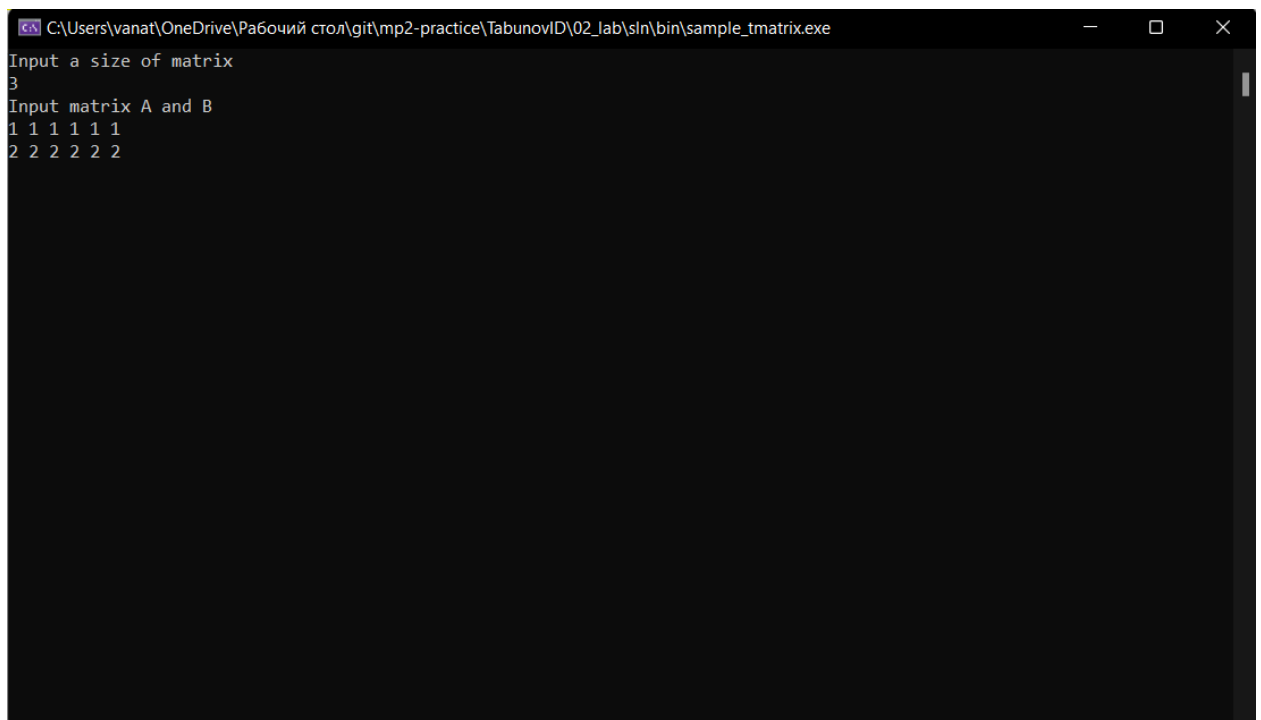
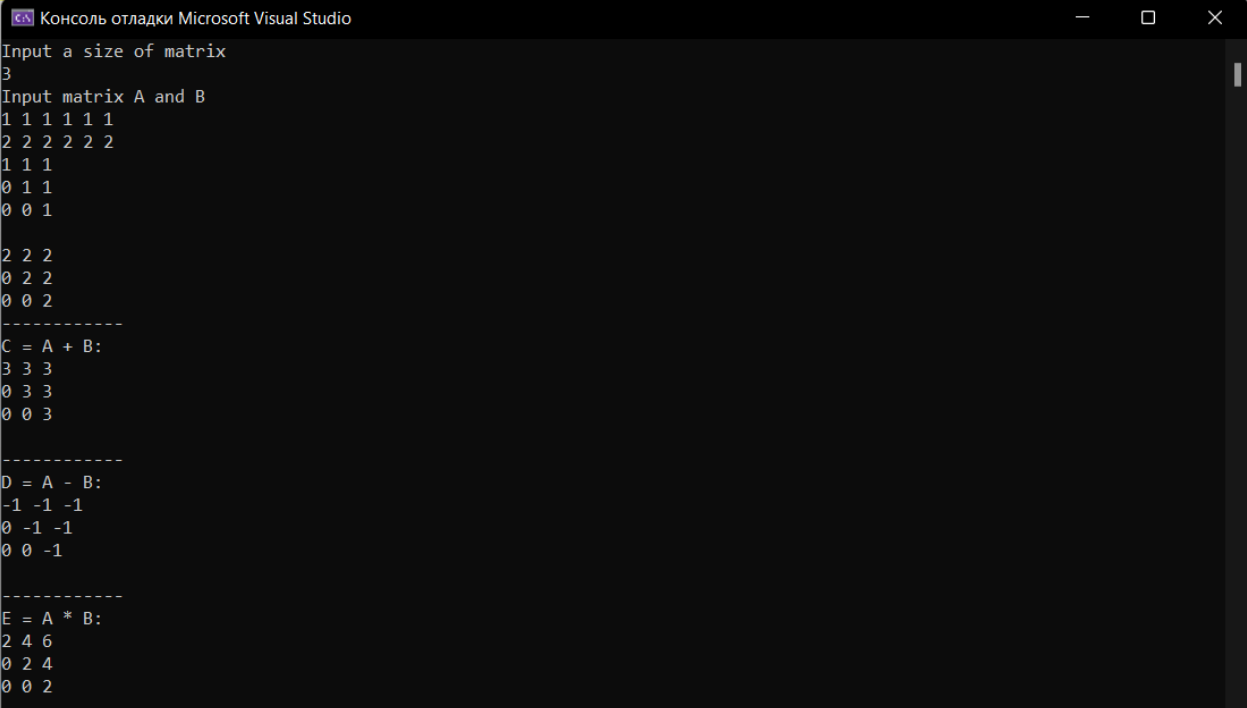


Рис. 5. Матрицы A, B

3. После ввода матриц будут выведены результаты соответствующих операций и функций (рис. 66).



```
Консоль отладки Microsoft Visual Studio
Input a size of matrix
3
Input matrix A and B
1 1 1 1 1 1
2 2 2 2 2 2
1 1 1
0 1 1
0 0 1

2 2 2
0 2 2
0 0 2
-----
C = A + B:
3 3 3
0 3 3
0 0 3

-----
D = A - B:
-1 -1 -1
0 -1 -1
0 0 -1

-----
E = A * B:
2 4 6
0 2 4
0 0 2
```

Рис. 6. Операции сложения, вычитания и умножения матриц



## 3 Руководство программиста

### 3.1 Описание алгоритмов

#### 3.1.1 Вектор

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

Вектор хранится в виде указателя на массив 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 будут равны нейтральному элементу типа данных.

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 векторов с элементами одинакового типа (складываются соответственные элементы векторов) или вектора и некоторого элемента того же типа (каждый элемент вектора отдельно складывается с элементом).

Пример:

Сложение векторов  $v_1 = \{1, 2, 3, 4\}$  и  $v_2 = \{1, 3, 5, 7\}$

$v_1 + v_2 = \{2, 5, 8, 11\}$

Сложение вектора  $v_1$ , с константой равной 5

$v_1 + 5 = \{6, 7, 8, 11\}$

##### **Операция вычитания**

Операция вычитания определена для векторов с элементами одинакового типа (вычитаются соответственные элементы векторов) или вектора и некоторого элемента того же типа (из каждого элемента вектора отдельно вычитается элемент).

Пример:

Разность векторов  $v_2 = \{1, 3, 5, 7\}$  и  $v_1 = \{1, 2, 3, 4\}$

$v_2 - v_1 = \{0, 1, 2, 3\}$

Вычитание из вектора  $v_1$  константы, равной 5

$v_1 - 5 = \{-4, -3, -2, -1\}$

##### **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:

Скалярное произведение векторов  $v2 = \{1, 3, 5, 7\}$  и  $v1 = \{1, 2, 3, 4\}$

$$v2 * v1 = 50$$

Произведение вектора  $v1$  с константой, равной 5

$$v1 * 5 = \{5, 10, 15, 20\}$$

### **Операция индексации**

Операция индексации предназначена для получения элемента вектора. Причем, если позиция будет меньше, чем стартовый индекс, то будет выведен нейтральный элемент для данного типа данных.

Пример:

$v1 = \{1, 2, 3, 4\}$ . Получение индекса 1 и 0 соответственно  $v1[1] = 2$ ,  $v1[0] = 1$

### **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:

$$v1 = \{1, 2, 3, 4\}, v2 = \{1, 3, 5, 7\}, v3 = \{1, 3, 5, 7\}$$

Сравнение векторов  $v1$  с  $v2$  и  $v2$  с  $v3$

$$(v1 == v2) = 0$$

$$(v2 == v3) = 1$$

### **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.

$$v2 = \{1, 3, 5, 7\}, v3 = \{1, 3, 5, 7\}$$

Сравнение векторов  $v2$  с  $v3$

$$(v2 != v3) = 0$$

## **3.1.2 Матрицы**

Структура данных верхнетреугольная матрица представляет собой вектор векторов. Она представлена как вектор, где каждый элемент является другим вектором, соответствующим строке матрицы и содержащим на 1 элемент меньше, чем предыдущий. Такая структура данных позволяет эффективно хранить разреженные матрицы и экономить память.

Над верхнетреугольной матрицей можно проводить следующие операции: умножение матрицы на число, сложение двух верхнетреугольных матриц, вычитание одной

верхнетреугольной матрицы из другой, проверка на равенство двух верхнетреугольных матриц, операции индексация для доступа к элементам матрицы.

### **Операция сложения**

Операция сложения определена для матриц с элементами одинакового типа (складываются соответственные элементы первой и второй матрицы). Она осуществляется за счёт работы с векторами. Мы берём наши соответствующие вектора (1-ый и 1-ый вектор, 2-ой и 2-ой и т. д.) и складываем их. Из получившихся векторов мы получаем нашу матрицу.

$$\text{Пример: } A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 3 & 3 & 3 \\ 0 & 6 & 6 \\ 0 & 0 & 9 \end{pmatrix}$$

### **Операция вычитания**

Операция вычитания определена для матриц с элементами одинакового типа (вычитаются соответствующие элементы второй матрицы из элементов первой матрицы). Она осуществляется за счёт работы с векторами. Мы берём наши соответствующие (1-ый и 1-ый вектор, 2-ой и 2-ой и т. д.) векторы и вычитаем вектор второй матрицы из вектора первой. Из получившихся векторов мы получаем нашу матрицу.

$$\text{Пример: } A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

$$A - B = \begin{pmatrix} -1 & -1 & -1 \\ 0 & -2 & -2 \\ 0 & 0 & -3 \end{pmatrix}$$

### **Операция умножения**

Операция умножения определена для матриц, которые представляют собой вектор векторов с элементами одинакового типа. Для того, чтобы получить элемент  $s_{ij}$  результирующей матрицы, нам нужно просуммировать произведения соответствующих элементов  $i$ -го вектора первой матрицы и  $j$ -го элемента вектора второй матрицы. Пример:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 2 & 6 & 12 \\ 0 & 8 & 20 \\ 0 & 0 & 18 \end{pmatrix}$$

### **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.

$$\text{Пример: } A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}, C = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

$$(A == B) = 0$$

$$(A == C) = 1$$

### Операция сравнения на неравенство

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

$$\text{Пример: } A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

$$(A != B) = 1$$

### Операция индексации

Операция индексации предназначена для получения элемента матрицы.

Элемент матрицы – вектор-строка, также можно вывести элемент матрицы по индексу, так как для вектора также перегружена операция индексации.

$$\text{Пример: } A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix}$$

$$A[0] = \{1, 1, 1\}$$

$$A[1][1] = 2$$

## 3.2 Описание программной реализации

### 3.2.1 Описание класса TVector

```
template <class T> class TVector
{
protected:
    int size;
    int start_index;
    T* pVec;
public:
    TVector(int s = 10, int index = 0);
    TVector(const TVector<T>& vec);
    ~TVector();
    int GetSize() const;
    int GetIndex() const;

    T& operator[](const int index);
    int operator==(const TVector<T>& v) const;
    int operator!=(const TVector<T>& v) const;

    TVector operator*(const T& v);
    TVector operator+(const T& v);
    TVector operator-(const T& v);

    TVector operator+(const TVector<T>& v);
    T operator*(const TVector<T>& v);
    TVector operator-(const TVector<T>& v);
```

```

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

template<typename T> friend std::ostream& operator<<(std::ostream& ostr,
const TVector<T>& v);
template<typename T> friend std::istream& operator>>(std::istream& istr,
TVector<T>& v);
};

```

Назначение: представление вектора.

Поля:

Size – количество элементов вектора.

Start\_Index – индекс первого необходимого элемента вектора.

\*pVec – память для представления элементов вектора.

Методы:

```
TVector(int s = 10, int index = 0);
```

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры: s – длина вектора, index – стартовый индекс.

```
TVector(const TVector<T>& vec);
```

Назначение: конструктор копирования.

Входные параметры: vec – экземпляр класса, на основе которого создаем новый объект.

```
~TVector();
```

Назначение: освобождение выделенной памяти.

```
int GetSize() const;
```

Назначение: получение размера вектора.

Выходные параметры: количество элементов вектора.

```
int GetIndex() const;
```

Назначение: получение стартового индекса.

Выходные параметры: стартовый индекс.

Операции:

```
T& operator[] (const int index);
```

Назначение: перегрузка операции индексации.

Входные параметры: index – индекс (позиция) элемента.

Выходные параметры: элемент, который находится на index позиции.

```
int operator==(const TVector<T>& v) const;
```

Назначение: оператор сравнения.

Входные параметры:  $v$  – экземпляр класса, с которым сравниваем.

Выходные параметры: 0 – если не равны, 1 – если равны.

```
int operator!=(const TVector<T>& v) const;
```

Назначение: оператор сравнения.

Входные параметры:  $v$  – экземпляр класса, с которым сравниваем.

Выходные параметры: 0 – если равны, 1 – если не равны.

```
TVector operator*(const T& v);
```

Назначение: оператор умножения вектора на значение.

Входные параметры:  $v$  – элемент, на который умножаем вектор.

Выходные параметры: экземпляр класса, элементы которого в  $v$  раз больше.

```
TVector operator+(const T& v);
```

Назначение: оператор сложения вектора и значения.

Входные параметры:  $v$  – элемент, с которым складываем вектор.

Выходные параметры: экземпляр класса, элементы которого на  $v$  больше.

```
TVector operator-(const T& v);
```

Назначение: оператор вычитания вектора и значения.

Входные параметры:  $v$  – элемент, который вычитаем из вектора.

Выходные параметры: экземпляр класса, элементы которого на  $v$  меньше.

```
TVector operator+(const TVector<T>& v);
```

Назначение: оператор сложения векторов.

Входные параметры:  $v$  – вектор, который суммируем.

Выходные параметры: экземпляр класса, равный сумме двух векторов.

```
T operator*(const TVector<T>& v);
```

Назначение: оператор умножения векторов.

Входные параметры:  $v$  – вектор, на который умножаем.

Выходные параметры: значение, равное скалярному произведению двух векторов.

```
TVector operator-(const TVector<T>& v);
```

Назначение: оператор разности двух векторов.

Входные параметры:  $v$  – вектор, который вычитаем.

Выходные параметры: экземпляр класса, равный разности двух векторов.

```
const TVector& operator=(const TVector<T>& v);
```

Назначение: оператор присваивания.

Входные параметры: `v` – экземпляр класса, который присваиваем.

Выходные параметры: ссылка на `(*this)`, уже присвоенный экземпляр класса.

```
template<typename T> friend std::ostream& operator>>(std::ostream& ostr,
                                                    const TVector<T>& v);
```

Назначение: оператор ввода вектора.

Входные параметры: `ostr` – поток ввода, `v` – ссылка на вектор, который выводим.

Выходные параметры: поток ввода.

```
template<typename T> friend std::istream& operator<<(std::istream& ostr,
                                                    TVector<T>& v);
```

Назначение: оператор вывода вектора.

Входные параметры: `ostr` – поток вывода, `v` – ссылка на вектор, который выводим.

Выходные параметры: поток вывода.

### 3.2.2 Описание класса TMatrix

```
template <typename T> class TMatrix : public TVector <TVector<T>>
{
public:
    TMatrix(int mn = 10);
    TMatrix(const TMatrix& m);
    TMatrix(const TVector <TVector<T>>& m);

    const TMatrix operator=(const TMatrix& m);
    int operator==(const TMatrix& m) const;
    int operator!=(const TMatrix& m) const;

    TMatrix operator+(const TMatrix& m);
    TMatrix operator-(const TMatrix& m);
    TMatrix operator*(const TMatrix& m);

    template<typename T> friend std::ostream& operator<<(std::ostream& ostr,
const TMatrix<T>& v);
    template<typename T> friend std::istream& operator>>(std::istream& istr,
TMatrix<T>& v);
};
```

Класс наследуется (тип наследования `public`) от класса `TVector<TVector<T>>`

Назначение: представление матрицы как вектор векторов.

Поля:

`Size` – размерность матрицы.

`Start_Index` – индекс первого необходимого элемента.

\*pVec – память для представления элементов матрицы.

Методы:

**TMatrix(int mn = 10);**

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры: mn – длина вектора (по умолчанию 10).

**TMatrix(const TMatrix& m);**

Назначение: конструктор копирования.

Входные параметры: m – экземпляр класса, на основе которого создаем новый объект.

**TMatrix(const TVector <TVector<T>>& m);**

Назначение: конструктор преобразования типов.

Входные параметры: m – ссылка на TVector<TVector<T>> - на объект, который преобразуем.

Операторы:

**const TMatrix operator=(const TMatrix& m);**

Назначение: оператор присваивания.

Входные параметры: m – экземпляр класса, который присваиваем.

Выходные параметры: ссылка на (\*this), уже присвоенный экземпляр класса.

**int operator==(const TMatrix& m) const;**

Назначение: оператор сравнения.

Входные параметры: m – экземпляр класса, с которым сравниваем.

Выходные параметры: 0 – если не равны, 1 – если равны.

**int operator!=(const TMatrix& m) const;**

Назначение: оператор сравнения.

Входные параметры: m – экземпляр класса, с которым сравниваем.

Выходные параметры: 0 – если равны, 1 – если не равны.

**TMatrix operator+(const TMatrix& m);**

Назначение: оператор сложения матриц.

Входные параметры: m – матрица, которую суммируем.

Выходные параметры: экземпляр класса, равный сумме двух матриц.

**TMatrix operator-(const TMatrix& m);**

Назначение: оператор вычитания матриц.



Входные параметры:  $m$  – матрица, которую вычитаем.

Выходные параметры: экземпляр класса, равный разности двух матриц.

```
TMatrix operator*(const TMatrix& m);
```

Назначение: оператор умножения матриц.

Входные параметры:  $m$  – матрица, которую умножаем.

Выходные параметры: экземпляр класса, равный произведению двух матриц.

```
template<typename T> friend std::istream& operator>>(std::istream& istr,  
                                                    TMatrix<T>& v);
```

Назначение: оператор ввода матрицы.

Входные параметры:  $istr$  – поток ввода,  $v$  – ссылка на матрицу, которую вводим.

Выходные параметры: поток ввода.

```
template<typename T> friend std::ostream& operator<<(std::ostream& ostr,  
                                                    const TMatrix<T>& v);
```

Назначение: оператор вывода матрицы.

Входные параметры:  $ostr$  – поток вывода,  $v$  – ссылка на матрицу, которую выводим.

Выходные параметры: поток вывода.

## **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](https://ru.wikipedia.org/wiki/Треугольная_матрица)].

# Приложения

## Приложение А. Реализация класса TVector

```
#ifndef __TVECTOR_H__
#define __TVECTOR_H__

#include <iostream>
#include <ostream>
#include <istream>
#include <iomanip>

template <class T> class TVector
{
protected:
    int size;
    int start_index;
    T* pVec;
public:
    TVector(int s = 10, int index = 0); //s = size
    TVector(const TVector<T>& vec);
    ~TVector();
    int GetSize() const;
    int GetIndex() const;

    T& operator[](const int index);
    int operator==(const TVector<T>& v) const;
    int operator!=(const TVector<T>& v) const;

    TVector operator*(const T& v); //multiplying by a number
    TVector operator+(const T& v);
    TVector operator-(const T& v);

    TVector operator+(const TVector<T>& v); // v1 + v2
    T operator*(const TVector<T>& v);
    TVector operator-(const TVector<T>& v);

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

    template<typename T> friend std::ostream& operator<<(std::ostream& ostr,
const TVector<T>& v);
    template<typename T> friend std::istream& operator>>(std::istream& istr,
TVector<T>& v);
};

//constructors:
template <typename T>
TVector<T> ::TVector(int s, int index)
{
    if (s < 0)
        throw - 1;
    else
        if (s == 0)
        {
            size = s;
            pVec = NULL;
        }
    else
    {
        start_index = index;
        size = s;
        pVec = new T[size];
    }
}
```

```

        for (int i = 0; i < size; i++)
            pVec[i] = 0;
    }
}

template <typename T>
TVector<T> ::TVector(const TVector<T>& vec)
{
    size = vec.size;
    start_index = vec.start_index;
    pVec = new T[size];
    for (int i = 0; i < size; i++)
    {
        pVec[i] = vec.pVec[i];
    }
}

//destruct
template <typename T>
TVector<T>::~~TVector()
{
    if (size > 0)
    {
        size = 0;
        delete[] pVec;
        pVec = NULL;
    }
}

//Get
template <typename T>
int TVector<T> :: GetSize() const
{
    return size;
}

template <typename T>
int TVector<T>::GetIndex() const
{
    return start_index;
}

//operators
template <typename T>
T& TVector<T> :: operator [] (const int index)
{
    if (index < 0 || index >= size)

        throw - 1;
    else
        return pVec[index];
}

template <typename T>
int TVector<T>::operator==(const TVector<T>& v) const
{
    if (size != v.size) {
        return 0;
    }

    for (int i = 0; i < size; i++) {
        if (pVec[i] != v.pVec[i]) {
            return 0;
        }
    }
}

```

```

    }
}
return 1;

}

template <typename T>
int TVector<T>::operator!=(const TVector<T>& v) const
{
    return !((*this) == v);
}

//operations vector with num
//vec * n
template <typename T>
TVector<T> TVector<T>::operator*(const T& v)
{
    TVector<T> res(size);
    for (int i = 0; i < size; i++)
        res[i] = (*this)[i] * v;
    return res;
}

// vec + const
template <typename T>
TVector<T> TVector<T>::operator+(const T& n)
{
    TVector<T> res(size);
    for (int i = 0; i < size; i++)
        res[i] = (*this)[i] + n;
    return res;
}

//vec - const
template <typename T>
TVector<T> TVector<T>::operator-(const T& n)
{
    TVector<T> res(size);
    for (int i = 0; i < size; i++)
        res[i] = (*this)[i] - n;
    return res;
}

//operations vec with vec
//vec + vec
template <typename T>
TVector<T> TVector<T>::operator+(const TVector<T>& v)
{
    if (size != v.size)
        throw "Can't accumulate vectors with different dimensions!";

    if (start_index != v.start_index)
        throw "Can't accumulate vectors with different indexes";

    TVector<T> tmp(*this);
    for (int i = 0; i < size; i++)
    {
        tmp.pVec[i] = pVec[i] + v.pVec[i];
    }
    return tmp;
}

//vec - vec
template <typename T>
TVector<T> TVector<T>::operator-(const TVector<T>& v)
{

```

```

    if (size != v.size)
        throw "Can't subtract vectors with different dimensions!";

    if (start_index != v.start_index)
        throw "Can't subtract vectors with different indexes";

    TVector<T> tmp(*this);
    for (int i = 0; i < size; i++)
    {
        tmp.pVec[i] = pVec[i] - v.pVec[i];
    }
    return tmp;
}

// scalar *
template <typename T>
T TVector<T>::operator*(const TVector<T>& v)
{
    if (size != v.size)
        throw "Can't scalar multiply vectors with different!";

    if (start_index != v.start_index)
        throw "Can't scalar multiply vectors with different indexes";

    T res = 0;
    for (int i = 0; i < size; i++)
    {
        res += pVec[i] * v.pVec[i];
    }
    return res;
}

// =
template <typename T>
const TVector<T>& TVector<T>::operator=(const TVector<T>& v)
{
    if (this == &v)
        return *this;

    if (size != v.size)
    {
        delete[] pVec;
        size = v.size;
        pVec = new T[size];
    }

    start_index = v.start_index;
    for (int i = 0; i < size; i++)
    {
        pVec[i] = v.pVec[i];
    }
    return *this;
}

// out
template <typename T>
std::ostream& operator<<(std::ostream& ostr, const TVector<T>& v)
{
    for (int i = 0; i < v.size; i++)
        ostr << std::setw(3) << v.pVec[i] << " ";
    return ostr;
}

//in
template <typename T>

```

```

std::istream& operator>>(std::istream& istr, TVector<T>& v)
{
    istr >> v.size;
    std::cout << "\nEnter the " << v.size << " coordinates: ";
    for (int i = 0; i < v.size; i++)
        istr >> v.pVec[i];
    return istr;
}
#endif

```

## Приложение Б. Реализация класса TMatrix

```

#ifndef __TMATRIX_H__
#define __TMATRIX_H__

#include "tvector.h"

template <typename T> class TMatrix : public TVector <TVector<T>>
{
public:
    TMatrix(int mn = 10);
    TMatrix(const TMatrix& m);
    TMatrix(const TVector <TVector<T>>& m);

    //operations
    const TMatrix operator=(const TMatrix& m);
    int operator==(const TMatrix& m) const;
    int operator!=(const TMatrix& m) const;

    TMatrix operator+(const TMatrix& m); // A + B
    TMatrix operator-(const TMatrix& m); // A - B
    TMatrix operator*(const TMatrix& m); // A * B

    template<typename T> friend std::ostream& operator<<(std::ostream& ostr,
const TMatrix<T>& v);
    template<typename T> friend std::istream& operator>>(std::istream& istr,
TMatrix<T>& v);

};
//constructors
template <typename T>
TMatrix<T>::TMatrix<T>(int mn) :TVector<TVector<T>>(mn)
{
    for (int i = 0; i < mn; i++)
    {
        pVec[i] = TVector<T>(mn - i, i);
    }
}

template<typename T>
TMatrix<T>::TMatrix<T>(const TMatrix& m):TVector<TVector<T>>(m) { };

template <typename T>
TMatrix<T>::TMatrix<T>(const TVector<TVector<T>>& m) :TVector<TVector<T>>(m) {
};

//oprations
// =
template<typename T>
const TMatrix<T> TMatrix<T>::operator=(const TMatrix<T>& m)
{
    return TVector<TVector<T>>::operator=(m);
}

```



```

// ==
template <typename T>
int TMatrix<T>::operator==(const TMatrix& m) const
{
    return TVector<TVector<T> >::operator==(m) ;
}

// !=
template <typename T>
int TMatrix<T>::operator!=(const TMatrix& m) const
{
    return TVector<TVector<T> >::operator!=(m) ;
}

// A + B
template<typename T>
TMatrix<T> TMatrix<T>::operator+(const TMatrix& m)
{
    if (size != m.size)
        throw "Can't accumulate matrix with different dimensions!";
    else
        return TVector<TVector<T> > :: operator+(m) ;
}

// A - B
template<typename T>
TMatrix<T> TMatrix<T>::operator-(const TMatrix& m)
{
    if (size != m.size)
        throw "Can't subtract matrix with different dimensions!";
    else
        return TVector<TVector<T> > :: operator-(m) ;
}

// A * B
template<typename T>
TMatrix<T> TMatrix<T>::operator*(const TMatrix& m)
{
    if (size != m.size)
        throw "Can't multiply matrix with different dimensions!";
    else
        size;
        TMatrix <T> res(size);
        for (int i = 0; i < size; i++)
            for (int j = i; j < size; j++)
            {
                for (int k = i; k <= j; k++)
                    res.pVec[i][j - i] += this->pVec[i][k - i] * m.pVec[k][j - k];
            }
        return res;
}

// in
template <typename T>
std::istream& operator>>(std::istream& istr, TMatrix<T>& m)
{
    istr >> m.size;
    std::cout << "\nEnter the " << m.size << " elements: ";
    for (int i = 0; i < m.size; i++)
        istr >> m.pVec[i];
    return istr;
}

// out

```

```

template <typename T>
std::ostream& operator<<(std::ostream& ostr, const TMatrix<T>& m)
{
    for (int i = 0; i < m.size; i++)
    {
        for (int j = 0; j < m.pVec[i].GetIndex(); j++)
            ostr << std::setw(3) << "0" << " ";
        ostr << m.pVec[i] << std::endl;
    }
    return ostr;
}
#endif

```

## Приложение В. Sample\_tmatrix

```

#include <iostream>

#include "tmatrix.h"

int main()
{
    TMatrix <double> A(3), B(3), C(3);

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3 - i; j++)
        {
            A[i][j] = 1 + i;
            B[i][j] = 2 + 2 * i;
            C[i][j] = A[i][j];
        }

    std::cout << "A:" << std::endl << A << std::endl;
    std::cout << "B:" << std::endl << B << std::endl;
    std::cout << "C:" << std::endl << C << std::endl;

    std::cout << "A + B:" << std::endl << A + B << std::endl;
    std::cout << "A - B:" << std::endl << A - B << std::endl;
    std::cout << "A * B:" << std::endl << A * B << std::endl;

    std::cout << "A == B ? " << (A == B) << std::endl;
    std::cout << "A != B ? " << (A != B) << std::endl;
    std::cout << "A == C ? " << (A == C) << std::endl;
    std::cout << std::endl;
    return 0;
}

```

## Приложение Г. Sample\_tvector

```

#include <iostream>
#include "tvector.h"

int main()
{
    TVector<int> v1(4,1);
    TVector<int> v2(4,1);
    TVector<int> v3(3, 2);
    TVector<int> v4(4, 1);

    for (int i = 0; i < 4; i++)

```

```

{
    v1[i] = i+1;
    v2[i] = i*2 +1;
    v4[i] = v2[i];
}

for (int i = 0; i < 3; i++)
{
    v3[i] = i*3 +1;
}

std::cout << "v1 = " << v1 << '\n';
std::cout << "v2 = " << v2 << '\n';
std::cout << "v3 = " << v3 << '\n';
std::cout << "v4 = " << v4 << '\n' << '\n';

//getSize
std::cout << "Size of v1 = v2 = " << v1.GetSize() << '\n';
std::cout << "Size of v3 = " << v3.GetSize() << '\n' << '\n';

//getIndex
std::cout << "Index of v1 = v2 = " << v1.GetIndex() << '\n';
std::cout << "Index of v3 = " << v3.GetIndex() << '\n' << '\n';

//element on position = index
std::cout << "Element of v2 on position 3 = " << v2 [3] << '\n' << '\n';

//==, !=
std::cout << "v1 == v2? " << (v1 == v2) << '\n';
std::cout << "v2 != v4? " << (v2 != v4) << '\n';
std::cout << "v2 == v4? " << (v2 == v4) << '\n' << '\n';

//vector and number
std::cout << "v1 = " << v1 << '\n' << '\n';

std::cout << "v1 * 5 = " << v1 * 5 << '\n';
std::cout << "v1 + 5 = " << v1 + 5 << '\n';
std::cout << "v1 - 5 = " << v1 - 5 << '\n' << '\n';

//vector and vector
std::cout << "v1 = " << v1 << '\n';
std::cout << "v2 = " << v2 << '\n' << '\n';

std::cout << "v1 + v2 = " << v1 + v2 << '\n';
std::cout << "v2 - v1 = " << v2 - v1 << '\n';
std::cout << "v1 scalar * v2 = " << v1 * v2 << '\n';

return 0;
}

```