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:

«Верхнетреугольные матрицы на шаблонах»

Выполнил: студент группы 3822Б1ФИ2
_____ /Миронов А. И./
Подпись

Проверил: к.т.н, доцент каф. ВВиСП
_____ / Кустикова В.Д./
Подпись

Нижний Новгород
2023

Содержание

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.

1 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.

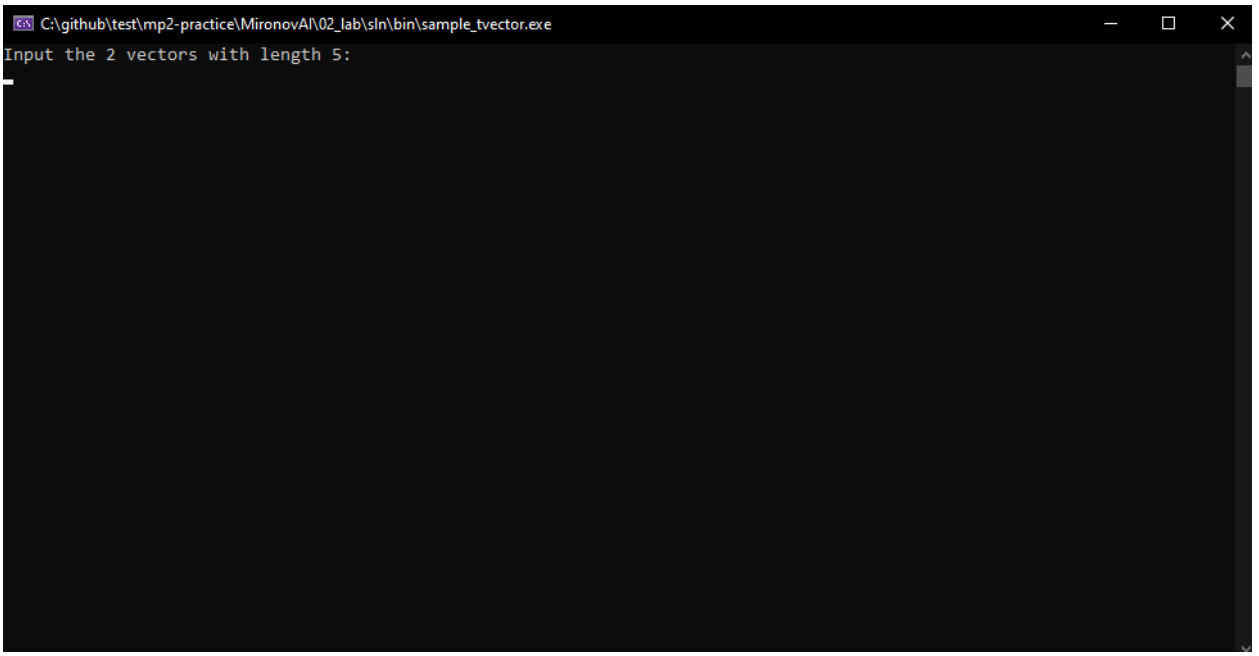
4. Проведение тестирования разработанных шаблонных классов на различных наборах данных для проверки их корректности и эффективности.

2 Руководство пользователя

2.1 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 5 (**Ошибка! Источник ссылки не найден.**).

Рис. 1. Основное окно программы



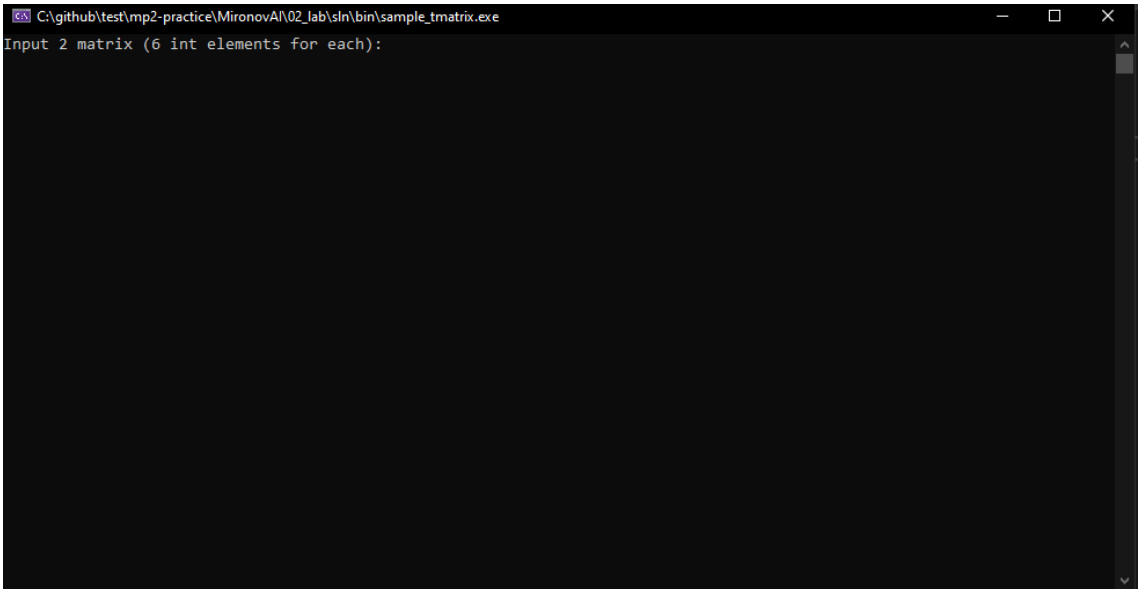
2. После ввода будет выведены результаты соответствующих операций и функций (рис 2).

Рис. 2. Результат тестирования функций класса `TVector`

2.2 Приложение для демонстрации работы матрицы

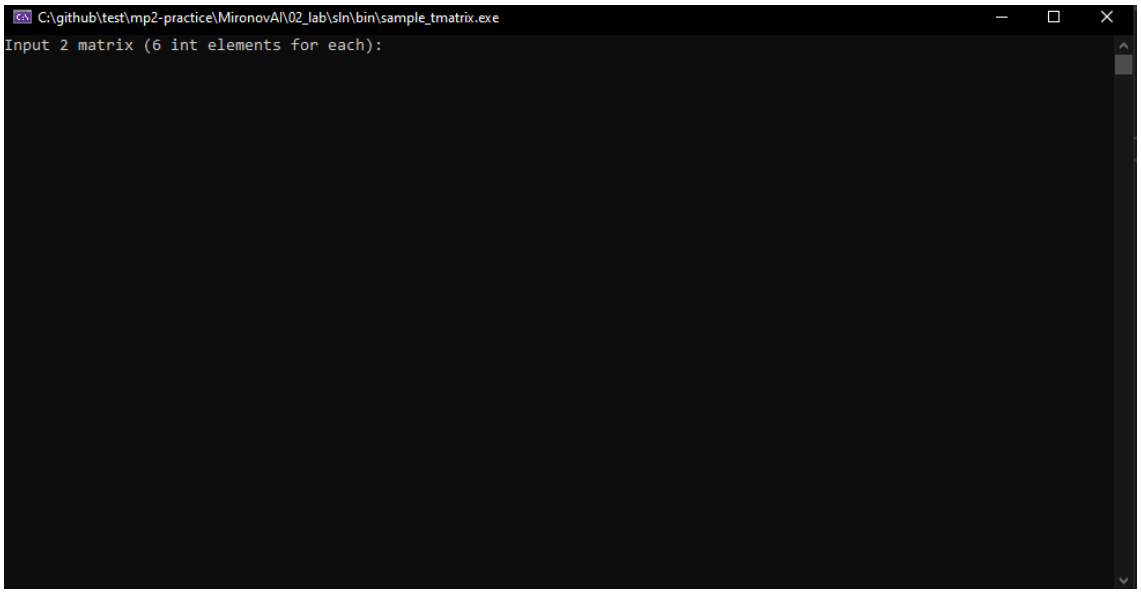
1. Запустите приложение с названием sample_tmatrix.exe. В результате появится окно, показанное ниже, вам будет предложено ввести 2 целочисленных верхнетреугольных матрицы 3 x 3 (вам нужно ввести 12 чисел) (рис 3).

Рис. 3. Основное окно программы



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

Рис. 4. Результат тестирования функций класса **`TMatrix`**



3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Вектор

1. Начало работы указано в пункте 2.1 этого документа
2. Описание методов и полей класса в пункте 3.2.1 этого документа

Программа алгоритм состоит из единственной функции **`void test_tvector()`**, которая сразу вызывается из основной функции **`int main()`**. Изначально необходимо алгоритм запрашивает пользователя ввести 2 вектора поля длинной 5 для демонстрации использования экземпляров класса (рис 5)

Рис. 5. Ввод данных алгоритма

```
TVector<int> v1(5), v2(5);
cout << "Input the 2 vectors with length 5:\n";
cin >> v1 >> v2;
```

3. После чего алгоритм последовательно выведет результат вызова соответствующих операций и методов класса (рис 6).

Рис. 6. Вывод результата работы алгоритма

```
cout << "First vector:\n" << v1 << endl;
cout << "Second vector:\n" << v2 << "\n\n";
cout << "v1+v2: " << v1 + v2 << endl;
cout << "v1-v2: " << v1 - v2 << endl;
cout << "v1*v2: " << v1 * v2 << endl;
cout << "v1-2: " << v1 - 2 << endl;
cout << "v2 + 2: " << v2 + 2 << endl;
cout << "v1.size(): " << v1.GetSize() << endl;
cout << "v1 == v2: " << (v1 == v2) << endl;
v1 = v2;
cout << "v1 == v2 after v1 = v2: " << (v1 == v2) << endl;
cout << "v1[1]: " << v1[1] << endl;
cout << "v2[0]: " << v2[0] << endl;
cout << "v1 != v2: " << (v1 != v2) << endl;

cout << "OK" << endl;
return;
```

3.1.2 Матрица

1. Начало работы указано в пункте 2.2 этого документа.
2. Описание методов и полей класса в пункте 3.2.2 этого документа

Программа алгоритм состоит из единственной функции **`void test_tmatrix()`**, которая сразу вызывается из основной функции **`int main()`**. Изначально необходимо алгоритм запрашивает пользователя ввести 2 верхнетреугольных матрицы, элементы которых должны быть целочисленным целым числом, для демонстрации использования экземпляров класса (рис 7)

Рис. 7. Ввод данных алгоритма

```
TMatrix<int> m1(3), m2(3);
cout << "Input 2 matrix (6 int elements for each): \n";
cin >> m1 >> m2;
```

3. После чего алгоритм последовательно выведет результат вызова соответствующих операций и методов класса (рис 8).

Рис. 8. Результат работы алгоритма

```
cout << "First mector:\n" << m1 << endl;
cout << "Second mector:\n" << m2 << "\n\n";
cout << "m1+m2:\n" << m1 + m2 << endl;
cout << "m1-m2:\n" << m1 - m2 << endl;
cout << "m1*m2:\n" << m1 * m2 << endl;
cout << "m1.size(): " << m1.GetSize() << endl;
cout << "m1 == m2: " << (m1 == m2) << endl;
m1 = m2;
cout << "m1 == m2 after m1 = m2: " << (m1 == m2) << endl;
cout << "m1[1]: " << m1[1] << endl;
cout << "m2[0]: " << m2[0] << endl;
cout << "m1 != m2: " << (m1 != m2) << endl;

cout << "OK" << endl;
return;
```

3.2 Описание программной реализации

3.2.1 Описание класса TVector

```
template <class ValType>
class TVector
{
private:
    int StartIndex;
protected:
  ValType *pVector;
  int Size;
  //int StartIndex;
public:
    TVector(int s = 10, int si = 0);
    TVector(const TVector& v);
    ~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: -

TVector(const Tvector<ValType>& v);

Назначение: конструктор копирования

Входные параметры:

v – экземпляр класса, на основе которого создаем новый объект

Выходные параметры: -

~TVector();

Назначение: деструктор

Входные параметры: -

Выходные параметры: -

int GetSize();

Назначение: получение размера вектора

Входные параметры: -

Выходные параметры: размер вектора (количество элементов)

int GetStartIndex();

Назначение: получение стартового индекса

Входные параметры: -

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 (*this) , уже 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

3.2.2 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>>.

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 – длинна вектора (по умолчанию 10)

Выходные параметры: -

TMatrix(const TMatrix &mt);

Назначение: конструктор копирования

Входные параметры:

mt – экземпляр класса, на основе которого создаем новый объект

Выходные параметры: -

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

Назначение: Конструктор преобразования типов

Входные параметры:

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

Выходные данные: -

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 (*this) , уже 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:

Out – 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

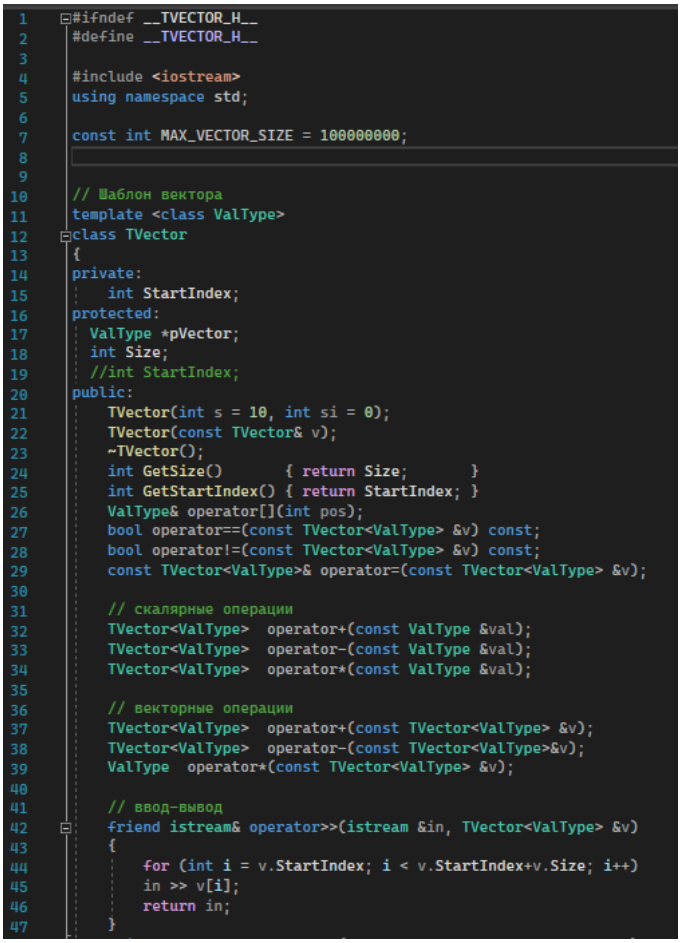
Литература

1. Википедия [https://ru.wikipedia.org/wiki/Треугольная_матрица].

Приложения

Приложение А. Реализация класса TVector

Рис. 9. Реализация класса TVector 1



```
#ifndef __TVECTOR_H__
#define __TVECTOR_H__

#include <iostream>
using namespace std;

const int MAX_VECTOR_SIZE = 100000000;


// Шаблон вектора
template <class ValType>
class TVector
{
private:
    int StartIndex;
protected:
  ValType *pVector;
  int Size;
  //int StartIndex;
public:
    TVector(int s = 10, int si = 0);
    TVector(const TVector& v);
    ~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)
    {
        for (int i = v.StartIndex; i < v.StartIndex+v.Size; i++)
        in >> v[i];
        return in;
    }
```

Рис. 10. Реализация класса TVector 2

```
48      friend ostream& operator<<(ostream &out,TVector<ValType> &v)
49      {
50        for (int i = 0; i < v.StartIndex + v.Size; i++)
51          out << v[i] << ' ';
52        return out;
53      }
54    };
55
56    template <class ValType>
57    TVector<ValType>::TVector(int s, int si):Size(s), StartIndex(si)
58    {
59        if (s <= 0 || s>MAX_VECTOR_SIZE)
60            throw "Incorrect size";
61        if (si < 0)
62            throw "You cannot start at negative index!";
63        pVector = new ValType[s]();
64    }
65
66    template <class ValType>
67    TVector<ValType>::TVector(const TVector<ValType> &v)
68    {
69      Size = v.Size;
70      StartIndex = v.StartIndex;
71      pVector = new ValType[Size];
72      std::copy(v.pVector, v.pVector + v.Size, pVector);
73    }
74    template <class ValType>
75    TVector<ValType>::~TVector()
76    {
77        delete[] pVector;
78    }
79
80    template <class ValType>
81    ValType& TVector<ValType>::operator[](int pos)
82    {
83        ValType x = ValType();
84
85        if (pos < 0 || pos>=MAX_VECTOR_SIZE)
86            throw "Wrong position";
87        if (pos < StartIndex)
88            return x;
89
90        if (pos - StartIndex < Size)
91            return pVector[pos - StartIndex];
92        else
93            throw "Acces Error";
94    }
```

Рис. 11. Реализация класса TVector 3

```
97    template <class ValType>
98    bool TVector<ValType>::operator==(const TVector<ValType>&v) const
99    {
100       //Нет перегрузки индексации для константного TVector
101       TVector tmp1(v), tmp2(*this);
102       //Реализация рассчитана на то, что в значимых элементах тоже могут быть нули
103       if (StartIndex + Size != v.StartIndex + v.Size)
104           return 0;
105
106
107       for (int i = min(StartIndex, v.StartIndex); i < StartIndex + Size; ++i)
108       {
109           if (tmp2[i] != tmp1[i])
110               return 0;
111       }
112       return 1;
113   }
114   /* ... */
133
134
135   template <class ValType>
136   bool TVector<ValType>::operator!=(const TVector<ValType>&v) const
137   {
138       return !(*this == v);
139   }
140
141
142   template <class ValType>
143   const TVector<ValType>& TVector<ValType>::operator=(const TVector<ValType>&v)
144   {
145       if (*this == v)
146           return *this;
147
148       if (Size != v.Size)
149       {
150           ValType* Tmp = new ValType[v.Size];
151           delete[] pVector;
152           pVector = Tmp;
153       }
154
155       Size = v.Size;
156       StartIndex = v.StartIndex;
157       std::copy(v.pVector, v.pVector + v.Size, pVector);
158   }
```

Рис. 12. Реализация класса TVector 4

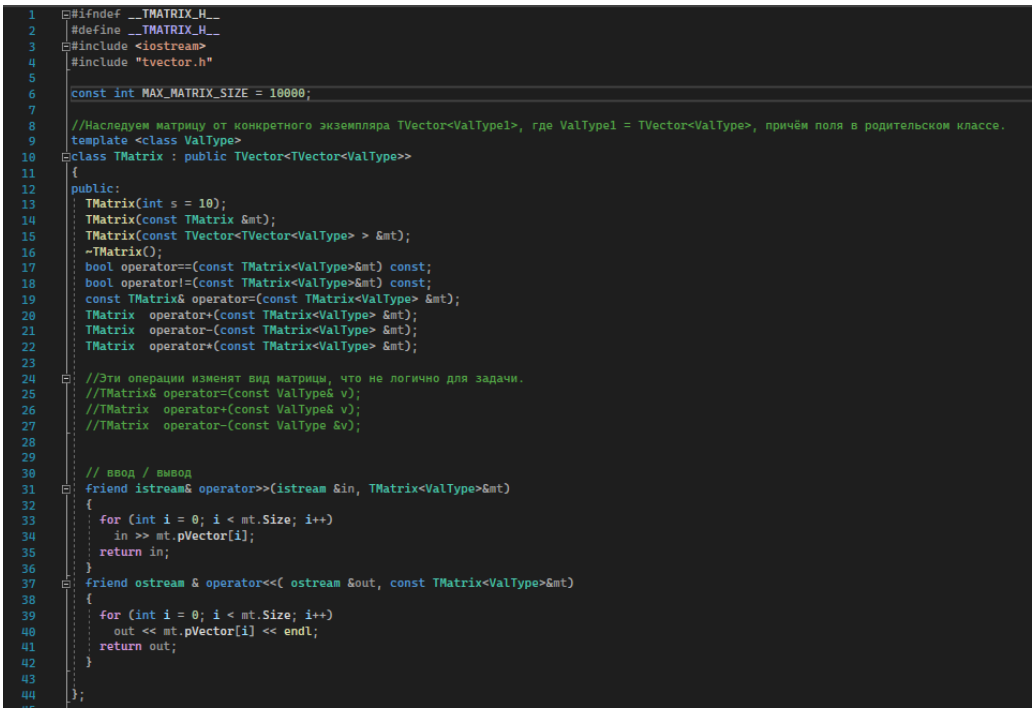
```
161  //После добавления числа к вектору, первые "нули" перестают быть нулями. Это учитывается. Другая реализация закомментирована
162  template <class ValType>
163  TVector<ValType> TVector<ValType>::operator+(const ValType &val)
164  {
165      TVector<ValType> A(Size+StartIndex);
166      for (int i = 0; i < A.Size; ++i)
167      {
168          A[i] = (*this)[i] + val;
169      }
170      return A;
171  }
172  /* ... */
184
185
186  template <class ValType>
187  TVector<ValType> TVector<ValType>::operator-(const ValType &val)
188  {
189      TVector<ValType> A(Size + StartIndex);
190      for (int i = 0; i < A.Size; ++i)
191      {
192          A[i] = (*this)[i] - val;
193      }
194      return A;
195  }
196
197
198  template <class ValType>
199  TVector<ValType> TVector<ValType>::operator*(const ValType &val)
200  {
201      TVector<ValType> A(Size , StartIndex);
202      for (int i = 0; i < A.Size; ++i)
203      {
204          A[i] = (*this)[i] * val;
205      }
206      return A;
207  }
208
209
210  template <class ValType>
211  TVector<ValType> TVector<ValType>::operator+(const TVector<ValType> &v)
212  {
213      TVector <ValType> B(max(v.Size, Size), min(v.StartIndex, StartIndex)), tmp(v);
214      for (int i = min(v.StartIndex, StartIndex); i < min(v.StartIndex, StartIndex) + max(v.Size, Size); ++i)
215      {
216          B[i] = (*this)[i] + tmp[i];
217      }
218      return B;
219  }
```

Рис. 13. Реализация класса TVector 5

```
221  /* ... */
237  template <class ValType>
238  TVector<ValType> TVector<ValType>::operator-(const TVector<ValType> &v)
239  {
240      TVector <ValType> B(max(v.Size, Size), min(v.StartIndex, StartIndex)), tmp(v);
241
242      for (int i = min(v.StartIndex, StartIndex); i < min(v.StartIndex, StartIndex) + max(v.Size, Size); ++i)
243      {
244          B[i] = (*this)[i] - tmp[i];
245      }
246      return B;
247  }
248
249  template <class ValType>
250  ValType TVector<ValType>::operator*(const TVector<ValType> &v)
251  {
252      ValType ans=ValType();
253      TVector <ValType> tmp(v);
254      for (int i = std::min(v.StartIndex, StartIndex); i < std::max(v.Size, Size); ++i)
255      {
256          ans = ans + (*this)[i] * tmp[i];
257      }
258      return ans;
259  }
260
261
262  #endif
```

Приложение Б. Реализация класса TMatrix

Рис. 14. Реализация класса TMatrix 1



```
#ifndef __TMATRIX_H__
#define __TMATRIX_H__
#include <iostream>
#include "tvector.h"

const int MAX_MATRIX_SIZE = 10000;

//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(int s = 10);
  TMatrix(const TMatrix &mt);
  TMatrix(const TVector<TVector<ValType> > &mt);
  ~TMatrix();
  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);

  //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;
  }

};
```

Рис. 15. Реализация класса TMatrix 2

```
template <class ValType>
TMatrix<ValType>::TMatrix(int s): TVector<TVector<ValType>>(s)
{
    if(s> MAX_MATRIX_SIZE)
        throw "Too Large Matrix";
    for (int i = 0; i < Size; ++i)
    {
        TVector<ValType> x(Size - i, i);
        pVector[i] = x;
    }
}

template <class ValType>
TMatrix<ValType>::TMatrix(const TMatrix<ValType> &mt):TVector<TVector<ValType>>(mt)
{
    for (int i = 0; i < Size; ++i)
    {
        pVector[i] = mt.pVector[i];
    }
}

template <class ValType>
TMatrix<ValType>::TMatrix(const TVector<TVector<ValType>> &mt):TVector<TVector<ValType>>(mt)
{
    if (mt.Size > MAX_MATRIX_SIZE)
    {
        throw "Allocation Error";
    }
}
template <class ValType>
TMatrix<ValType>::~TMatrix()
{
    for (int i = 0; i < Size; i++)
    {
        delete[] this[i];
    }
}

template <class ValType>
bool TMatrix<ValType>::operator==(const TMatrix<ValType> &mt) const
{
    for (int i = 0; i < Size; ++i)
    {
        if (pVector[i] != mt.pVector[i])
            return 0;
    }
    return 1;
}
```

Рис. 16. Реализация класса TMatrix 3

```
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)
{
    if (*this == mt)
        return *this;

    if (Size != mt.Size)
    {
        TVector<ValType>* Tmp  = new TVector<ValType>[mt.Size];
        delete[] pVector;
        pVector = Tmp;
    }

    Size = mt.Size;
    std::copy(mt.pVector, mt.pVector + mt.Size, pVector);
    return *this;
}

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

    TMatrix<ValType> A(std::max(Size, mt.Size)), not_const_mt(mt);
    for (int i = 0; i < max(Size, mt.Size); ++i)
    {
        TVector<ValType> tmp;
        if (i >= not_const_mt.Size)
            tmp = (*this)[i];
        else if (i >= Size)
            tmp = not_const_mt[i];
        else
            tmp = (*this)[i] + not_const_mt[i];
        A[i] = tmp;
    }
    return A;
}
```

Рис. 17. Реализация класса TMatrix 4

```

template <class ValType>
TMatrix<ValType> TMatrix<ValType>::operator-(const TMatrix<ValType> &mt)
{
    TMatrix<ValType> A(std::max(Size, mt.Size)), not_const_mt(mt);
    for (int i = 0; i < max(Size, mt.Size); ++i)
    {
        TVector<ValType> tmp;
        if (i >= not_const_mt.Size)
            tmp = (*this)[i];
        else if (i >= Size)
            tmp =   not_const_mt[i]*(-1);
        else
            tmp = (*this)[i] - not_const_mt[i];
        A[i] = tmp;
    }
    return A;
}
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

```