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:

«Вектора и матрицы»

Выполнил(а): студент(ка) группы 3822Б1ФИ2
_____ / Чижов М.А./
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 6

3 Руководство программиста 7

3.1 Описание алгоритмов 7

3.1.1 Вектора 7

3.1.2 Матрицы 9

3.2 Описание программной реализации 12

3.2.1 Описание класса TVector 12

3.2.2 Описание класса TMatrix 16

Заключение 19

Литература 20

Приложения 21

Приложение А. Реализация класса TVector 21

Приложение Б. Реализация класса TMatrix 23

Введение

Векторы и матрицы в программировании являются важными инструментами для работы с числовыми данными. В языке программирования C++ существуют различные способы работы с векторами и матрицами, которые позволяют эффективно решать различные задачи.

Матрицы в C++ можно представить в виде вектора векторов (вложенных векторов). Для работы с матрицами в C++ можно использовать встроенные типы данных, такие как массив или обычный вектор. Однако использование вектора векторов является более гибким и позволяет легко изменять размеры строк и столбцов матрицы.

Для программистов матрицы полезны во множестве задач: от обработки изображений и аудио до работы с графическими данными и алгоритмами машинного обучения. Особенно важно понимание и использование матриц в таких областях, как компьютерная графика и обработка сигналов.

Одним из преимуществ матриц является их способность представлять сложные данные в простой и удобной форме. Каждый элемент матрицы имеет свои координаты, что позволяет быстро и удобно обращаться к нужным данным. Кроме того, матрицы предоставляют специализированные операции для работы с данными, такие как умножение, сложение и транспонирование, что делает их неотъемлемой частью многих алгоритмов и программ.

1 Постановка задачи

Цель – реализовать структуру хранения векторов и верхнетреугольной матрицы.

Задачи:

1. Реализовать классы для работы с векторами и верхнетреугольными матрицами
2. Написать следующие операции для работы с векторами: прибавление числа к вектору, вычитание числа из вектора, умножение вектора на число.Сложение, вычитание, умножение векторов. Операции присваивания и сравнение на равенство или неравенство.
3. Добавить вспомогательные операции получения размера и стартового индекса вектора.
4. Написать следующие операции для работы с матрицами: Сложение, вычитание и умножение матриц.Операции присваивания и сравнение на равенство или неравенство.

2 Руководство пользователя

2.1 Приложение для демонстрации работы векторов

1. Запустите приложение с названием sample_vec.exe. В результате появится окно, показанное ниже (рис. 1).

Рис. 1. Основное окно программы

2. Введите целое число для размера вектора, а после заполните первый вектор целыми числами, количество которых равно размеру вектора (рис. 2).

Рис. 2. Ввод длины и первого вектора

3. Введите второй вектор, указанного выше размера (рис. 3).

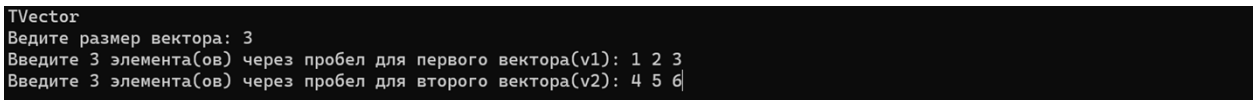

Рис. 3. Ввод второго вектора

4. После ввода второго вектора выполнится операция присваивания, а после нужно будет ввести третий вектор, указанного выше размера (рис. 4).

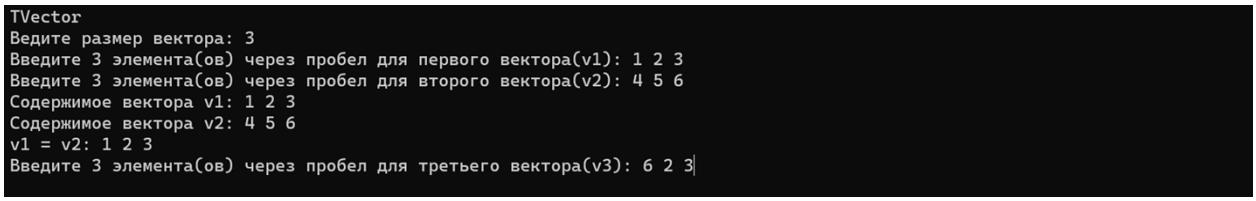

Рис. 4. Ввод третьего вектора

5. После ввода третьего вектора появится окно с результатами операций ().

Рис. 5. Результат операций

2.2 Приложение для демонстрации работы матриц

1. Запустите приложение с названием sample_matrix.exe. Введите размер матрицы (рис. 6).

Рис. 6. Основное окно программы

2. Введите первую матрицу, как показано ниже (рис. 7).

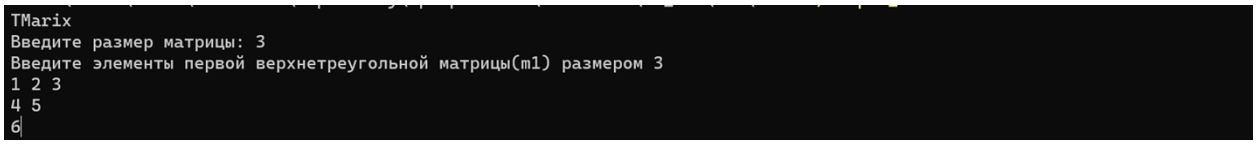

Рис. 7. Ввод первой матрицы

3. Введите вторую матрицу (рис. 8).

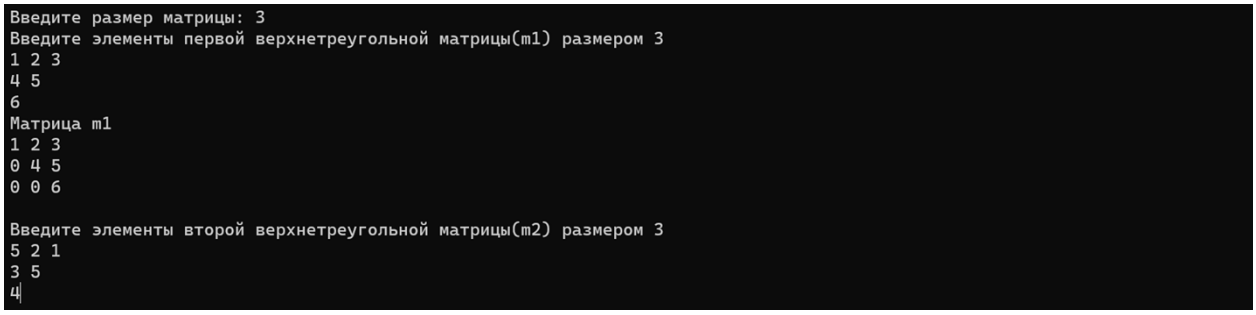

Рис. 8. Ввод второй матрицы

4. После ввода второй матрицы появится окно с выполненными операциями (рис. 9).

Рис. 9. Результат операций

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Вектора

Класс TVector представляет собой шаблонный. Для создания экземпляра класса TVector необходимо указать его размер и начальный индекс. По умолчанию размер устанавливается как 5, а начальный индекс – 0. Класс TVector также поддерживает ряд операций, включая сравнение векторов на равенство и неравенство, умножение вектора на скалярное значение, сложение и вычитание скаляра из вектора, а также сложение, вычитание и скалярное произведение с другим вектором.

Операция получения размера вектора

5	6	1	7	2

Результат:

5 – длина вектора.

Операция получения стартового индекса

5	6	1	7	2

Результат:

0 – стартовый индекс.

Операция сравнение на равенство

5	6	1	7	2
5	6	1	7	2

Результат:

1 – векторы равны

5	6	1	7	2
5	2	1	7	2

Результат:

0 – векторы не равны

Операция сравнение на неравенство

5	6	1	7	2
5	6	3	7	2

Результат:

1 – векторы не равны.

5	6	1	7	2
5	6	1	7	2

Результат:

0 – векторы равны.

Операция присваивания

5	6	1	7	2
3	6	3	1	2

В результате присваивания первого (верхнего) вектора второму (нижнему) вектору, получится следующий вектор:

5	6	1	7	2

Операция сложения вектора со скаляром

5	6	1	7	2

скаляр = 2

Результат:

7	8	3	9	4

Операция вычитания из вектора скаляра

5	6	3	7	2

скаляр = 2

Результат:

3	4	1	5	0

Операция умножения вектора на скаляр

1	3	1	4	2

скаляр = 2

Результат:

2	6	2	8	4

Операция сложения векторов

5	6	1	7	2
3	1	5	2	6

Результат:

8	7	6	9	8

Операция вычитания векторов

5	6	4	7	6
3	1	1	2	2

Результат:

2	5	3	5	4

Операция умножения векторов

5	6	4	7	6
3	1	1	2	2

Результат: 51

3.1.2 Матрицы

Класс TMatrix представляет собой шаблонный класс. Класс TMatrix наследует функциональность класса TVector<TVector>, что позволяет удобно оперировать с двумерными массивами. Для создания экземпляра класса TMatrix необходимо указать размерность матрицы при вызове конструктора. При этом можно также передать значения по умолчанию для инициализации элементов матрицы. Класс TMatrix также поддерживает операции присваивания, сравнения матриц на равенство и неравенство, а также выполнение арифметических операций над матрицами, таких как сложение, вычитание и умножение.

Операция присваивания

1	3	2
0	2	2
0	0	1

3	5	4
0	1	5
0	0	4

В результате присваивания первой (верхней) матрицы второй (нижней) матрицы, получится следующая матрица:

1	3	2

0	2	2
0	0	1

Операция сравнения на равенство

1	3	2
0	2	2
0	0	1

3	5	4
0	1	5
0	0	4

Результат:

0 – матрицы не равны

1	3	2
0	2	2
0	0	1

1	3	2
0	2	2
0	0	1

Результат:

1 – матрицы равны

Операция сравнения на неравенство

1	3	2
0	2	2
0	0	1

3	5	4
0	1	5
0	0	4

Результат:

1 – матрицы не равны

1	3	2

0	2	2
0	0	1

1	3	2
0	2	2
0	0	1

Результат:

0 – матрицы равны

Операция сложения матриц

1	3	2
0	2	2
0	0	1

3	5	4
0	1	5
0	0	4

Результат:

4	8	6
0	3	7
0	0	5

Операция вычитания матриц

3	5	4
0	2	5
0	0	7

1	3	2
0	1	1
0	0	4

Результат:

2	2	2
0	1	4

0	0	3

Операция умножения матриц

3	5	4
0	2	5
0	0	7

1	3	2
0	1	1
0	0	4

Результат:

3	14	27
0	2	22
0	0	28

3.2 Описание программной реализации

3.2.1 Описание класса TVector

```
template <typename T>
class TVector
{
protected:
	T* vec;
	int size;
	int start_ind;
public:
	TVector(int _size=5, int _start=0);
	TVector(const TVector<T>& obj);
	virtual ~TVector();

	int GetSize() const;
	int GetStart() const;
	T& operator [](const int index);

	int operator ==(const TVector<T>& obj) const;
	int operator !=(const TVector<T>& obj) const;
	TVector<T> operator *(const T& val);
	TVector<T> operator +(const T& val);
	TVector<T> operator -(const T& val);

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

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

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

        friend istream& operator >>(istream& in, TVector<T>& obj)
        {
              for (int i = 0; i < obj.size; i++)
              {
                    in >> obj.vec[i];
                    in.ignore();
              }
              return in;
        };
        friend ostream& operator <<(ostream& out, const TVector<T>& obj)
        {
              for (int i = 0; i < obj.start_ind; i++)
                    out << "0 ";
              for (int i = 0; i < obj.size; i++)
                    out << obj.vec[i] << " ";
              return out;
        };
};
```

Назначение: представление вектора.

Поля:

vec – указатель на массив типа **T**.

size – размер вектора.

start_ind – стартовый индекс вектора.

Методы:

```
TVector(int _size=5, int _start=0);
```

Назначение: создает объект класса **TVector** с указанным размером и начальным индексом.

Входные параметры:

_size – размер вектора (по умолчанию 5).

_start – начальный индекс (по умолчанию 0).

Выходные параметры: отсутствуют.

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

Назначение: создает копию существующего объекта класса **TVector**.

Входные параметры **obj** – объект класса **TVector**, который нужно скопировать.

Выходные параметры: отсутствуют.

```
virtual ~TVector();
```

Назначение: освобождает выделенную память для вектора.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: возвращает размер вектора.

Входные параметры: отсутствуют.

Выходные параметры: размер вектора.

```
int GetStart() const;
```

Назначение: возвращает начальный индекс вектора.

Входные параметры: отсутствуют.

Выходные параметры: начальный индекс вектора.

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

Назначение: позволяет получить доступ к элементу вектора по его индексу.

Входные параметры: **index** – индекс элемента.

Выходные параметры: элемент вектора типа **T**.

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

Назначение: сравнивает два вектора на равенство.

Входные параметры: **obj** – объект класса **TVector** для сравнения.

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

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

Назначение: Сравнивает два вектора на неравенство.

Входные параметры: **obj** – объект класса **TVector** для сравнения.

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

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

Назначение: присваивание вектора.

Входные параметры: **obj** – объект класса **TVector**, который нужно присвоить.

Выходные параметры: ссылка на текущий объект класса **TVector**.

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

Назначение: перемножает каждый элемент вектора на указанное значение.

Входные параметры: **val** – значение, на которое нужно умножить элементы вектора.

Выходные параметры: вектор типа **TVector**, содержащий результат умножения.

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

Назначение: прибавляет указанное значение к каждому элементу вектора.

Входные параметры: **val** – значение, которое нужно прибавить к элементам вектора.

Выходные параметры: вектор типа **TVector**, содержащий результат сложения.

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

Назначение: вычитает указанное значение из каждого элемента вектора.

Входные параметры: **val** – значение, которое нужно вычесть из элементов вектора.

Выходные параметры: вектор типа **TVector**, содержащий результат вычитания.

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

Назначение: сложение векторов.

Входные параметры: **obj** – объект класса **TVector**, который нужно прибавить.

Выходные параметры: вектор типа **TVector**, содержащий результат сложения.

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

Назначение: вычитание векторов.

Входные параметры: **obj** – объект класса **TVector**, который нужно вычесть.

Выходные параметры: вектор типа **TVector**, содержащий результат вычитания.

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

Назначение: умножение векторов.

Входные параметры: **obj** – объект класса **TVector**, который нужно вычесть.

Выходные параметры: результат умножения в виде значения типа **T**.

```
friend istream& operator >>(istream& in, TVector<T>& obj);
```

Назначение: оператор ввода для класса **TVector**.

Входные параметры:

istr – ссылка на объект типа **istream**, который представляет входной поток.

obj – ссылка на объект типа **TVector**, в который будут считываться значения.

Выходные параметры: ссылка на объект типа **istream**.

```
friend ostream& operator <<(ostream& out, const TVector<T>& obj);
```

Назначение: оператор вывода для класса **TVector**.

Входные параметры:

ostr – ссылка на объект типа **ostream**, который представляет выходной поток.

obj – ссылка на объект типа **TVector**, который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

3.2.2 Описание класса TMatrix

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

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

      const TMatrix<T> operator=(const TMatrix<T>& m);
      int operator==(const TMatrix<T>& m) const;
      int operator!=(const TMatrix<T>& m) const;

      TMatrix operator +(const TMatrix<T>& m);
      TMatrix operator -(const TMatrix<T>& m);
      TMatrix operator *(const TMatrix<T>& m);

      friend istream& operator >>(istream& in, TMatrix<T>& m)
      {
            for (int i = 0; i < m.GetSize(); i++)
                  in >> m.vec[i];

            return in;
      };
      friend ostream& operator <<(ostream& out, TMatrix<T>& m)
      {
            for (int i = 0; i < m.GetSize(); i++)
                  out << m.vec[i] << endl;

            return out;
      };
};
```

Назначение: представление верхнетреугольной матрицы

Методы:

```
TMatrix(int mp);
```

Назначение: создает объект **TMatrix** заданного размера.

Входные параметры: **mp** – размер матрицы.

Выходные параметры: созданный объект **TMatrix.**

```
TMatrix(const TMatrix<T>& m);
```

Назначение: создает копию объекта **TMatrix**.

Входные параметры: **m** - объект **TMatrix**, который нужно скопировать.

Выходные параметры: отсутствуют.

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

Назначение: создает объект **TMatrix** на основе векторов.

Входные параметры: **m** – ссылка на объект **TMatrix**.

Выходные параметры: созданный объект **TMatrix.**

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

Назначение: позволяет получить доступ к элементу матрицы по его индексу.

Входные параметры: **index** – индекс объекта.

Выходные параметры: элемент вектора типа **TVector**.

```
int operator==(const TMatrix<T>& m) const;
```

Назначение: сравнивает две матрицы на равенство.

Входные параметры: **m** – ссылка на объект **TMatrix**.

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

```
int operator!=(const TMatrix<T>& m) const;
```

Назначение: сравнивает две матрицы на неравенство

Входные параметры: **m** – ссылка на объект **TMatrix**.

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

```
const TMatrix<T> operator=(const TMatrix<T>& m);
```

Назначение: присваивание матрицы.

Входные параметры: **m** – ссылка на объект **TMatrix**.

Выходные параметры: ссылка на текущий объект класса **TMatrix**.

```
TMatrix operator +(const TMatrix<T>& m);
```

Назначение: сложение матриц.

Входные параметры: **m** – ссылка на объект **TMatrix**.

Выходные параметры: новый объект **TMatrix,** являющийся результатом сложения двух матриц.

```
TMatrix operator -(const TMatrix<T>& m);
```

Назначение: вычитание матриц.

Входные параметры: **m** – ссылка на объект **TMatrix**.

Выходные параметры: новый объект **TMatrix**, являющийся результатом вычитания двух матриц.

```
TMatrix operator *(const TMatrix<T>& m);
```

Назначение: умножение двух матриц.

Входные параметры: **m** – ссылка на объект **TMatrix**.

Выходные параметры: новый объект **TMatrix**, являющийся результатом умножения двух матриц.

```
friend istream& operator >>(istream& in, TMatrix<T>& m);
```

Назначение: оператор ввода для класса **TMatrix**.

Входные параметры:

in – ссылка на объект типа **istream**, который представляет входной поток.

m– ссылка на объект типа **TMatrix**, в который будут считываться значения.

Выходные параметры: ссылка на объект типа **istream**.

```
friend ostream& operator <<(ostream& out, TMatrix<T>& m);
```

Назначение: оператор вывода для класса **TMatrix**.

Входные параметры:

out – ссылка на объект типа **ostream**, который представляет выходной поток.

m – ссылка на объект типа **TMatrix**, который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

Заключение

В результате разработки классов TVector и TMatrix была создана структура данных, позволяющая эффективно работать с векторами и матрицами.

В результате разработки класса вектора мы получили возможность эффективно оперировать векторами и выполнять различные операции над ними, включая сложение, вычитание, умножение на скаляр, нахождение скалярного произведения, сложение и вычитание векторов, сравнение на равенство и неравенство. Класс вектора позволяет удобно хранить и обрабатывать числовые значения вектора и предоставляет удобные методы для доступа и модификации его элементов.

Также, благодаря разработке класса матрицы, мы получили возможность оперировать не только с векторами, но и с матрицами. Класс матрицы позволяет хранить и обрабатывать многомерную информацию и предоставляет методы для выполнения различных операций, включая сложение матриц, вычитание, умножение матриц и сравнение на равенство и неравенство.

Литература

1. Применение матриц в программировании [https://apexwiki.ru/primenenie-matric-v-programmirovanii/]
2. Векторы и матрицы c++ [https://qaa-engineer.ru/vektory-i-matriczy-c/]

Приложения

Приложение А. Реализация класса TVector

```
template <typename T>
TVector<T>::TVector(int _size, int _start)
{
	if (_size <= 0 || _start < 0)
		throw "Error";
	size = _size;
	start_ind = _start;
	vec = new T[size];
};
template <typename T>
TVector<T>::TVector(const TVector<T>& obj)
{
	size = obj.size;
	start_ind = obj.start_ind;
	vec = new T[size];
	for (int i = 0; i < size; i++)
		vec[i] = obj.vec[i];
};
template <typename T>
TVector<T>::~TVector()
{
	if (size > 0)
		delete[] vec;
};
template <typename T>
int TVector<T>::GetSize() const
{
	return size;
}
template <typename T>
int TVector<T>::GetStart() const
{
	return start_ind;
}
template <typename T>
T& TVector<T>::operator [](const int index)
{
	if (index < 0 || index >= size + start_ind)
		throw "error";
	if (index < start_ind)
		throw "error";

	return vec[index - start_ind];
}
template <typename T>
int TVector<T>::operator==(const TVector<T>& obj) const
{
	if (size != obj.size || start_ind != obj.start_ind)
		return false;
	if (this == &obj)
		return 1;
	for (int i = 0; i < size; i++)
		if (vec[i] != obj.vec[i])
			return false;
	return true;
};
template <typename T>
int TVector<T>::operator!=(const TVector<T>& obj) const
```

```
{
	return !((*this) == obj);
};
template <typename T>
TVector<T> TVector<T>::operator *(const T& val)
{
	TVector<T> result(size, start_ind);
	for (int i = 0; i < size; i++)
		result.vec[i] = vec[i] * val;
	return result;
}
template <typename T>
TVector<T> TVector<T>::operator +(const T& val)
{
	TVector<T> result(size, start_ind);
	for (int i = 0; i < size; i++)
		result.vec[i] = vec[i] + val;
	return result;
}
template <typename T>
TVector<T> TVector<T>::operator -(const T& val)
{
	TVector<T> result(size, start_ind);
	for (int i = 0; i < size; i++)
		result.vec[i] = vec[i] - val;
	return result;
}

template <typename T>
TVector<T> TVector<T>::operator +(const TVector<T>& obj)
{
	if (size != obj.size || start_ind != obj.start_ind)
		throw "Error";
	TVector<T> res(size, start_ind);
	for (int i = 0; i < size; i++)
		res.vec[i] = vec[i] + obj.vec[i];
	return res;
};
template <typename T>
TVector<T> TVector<T> ::operator -(const TVector<T>& obj)
{
	if (size != obj.size || start_ind != obj.start_ind)
		throw "Error";
	TVector<T> res(size, start_ind);
	for (int i = 0; i < size; i++)
		res.vec[i] = vec[i] - obj.vec[i];
	return res;
};
template <typename T>
T TVector<T>::operator *(const TVector<T>& obj)
{
	if (size != obj.size || start_ind != obj.start_ind)
		throw "Error";
	T res = 0;
	for (int i = 0; i < size; i++)
		res += vec[i] * obj.vec[i];
	return res;
};

template <typename T>
const TVector<T>& TVector<T>::operator=(const TVector<T>& obj)
{
	if (*this == obj)
```

```
            return (*this);
      if (size != obj.size || start_ind != obj.start_ind)
      {
            delete[] vec;
            size = obj.size;
            start_ind = obj.start_ind;
            vec = new T[size];
      }
      for (int i = 0; i < size; i++)
            vec[i] = obj.vec[i];
      return (*this);
};
friend istream& operator >>(istream& in, TVector<T>& obj)
{
      for (int i = 0; i < obj.size; i++)
      {
            in >> obj.vec[i];
            in.ignore();
      }
      return in;
};
friend ostream& operator <<(ostream& out, const TVector<T>& obj)
{
      for (int i = 0; i < obj.start_ind; i++)
            out << "0 ";
      for (int i = 0; i < obj.size; i++)
            out << obj.vec[i] << " ";
      return out;
};
```

Приложение Б. Реализация класса TMatrix

```
template <typename T>
TMatrix<T> :: TMatrix(int mn) : TVector<TVector<T>> (mn)
{
      if (mn <= 0)
            throw "ERROR: matrix size less than 1";
      for (int i = 0; i < mn; i++)
            (*this)[i] = TVector<T>(mn - i, i);
}

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

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

template <typename T>
TVector<T>& TMatrix<T>::operator[](const int index) {
      return TVector<TVector<T>>::operator[](index);
}

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<T>& m) const
{
```

```
      return TVector<TVector<T>>::operator==(m);
};

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

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

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

template <typename T>
TMatrix<T> TMatrix<T>::operator*(const TMatrix<T>& m)
{
      if      (TVector<TVector<T>>::GetSize()      !=      m.GetSize()      ||
TVector<TVector<T>>::GetStart() != m.GetStart())
            throw "ERROR: matrix sizes do not match";

      TMatrix<T> result(TVector<TVector<T>>::GetSize());
      for (int i = 0; i < TVector<TVector<T>>::GetSize(); i++)
            for      (int      j      =      result[i].GetStart();      j      <
TVector<TVector<T>>::GetSize(); j++)
                  result[i][j] = T(NULL);

      for (int i = 0; i < TVector<TVector<T>>::GetSize(); i++)
      {
            for      (int      j      =      result[i].GetStart();      j      <
TVector<TVector<T>>::GetSize(); j++)
            {
                  T sum = 0;
                  for (int k = result[i].GetStart(); k <= j; k++) {
                        sum += (*this)[i][k] * m.vec[k][j];
                  }
                  result.vec[i][j] = sum;
            }
      }
      return result;
};
friend istream& operator >>(istream& in, TMatrix<T>& m)
{
      for (int i = 0; i < m.GetSize(); i++)
            in >> m.vec[i];

      return in;
};
friend ostream& operator <<(ostream& out, TMatrix<T>& m)
{
      for (int i = 0; i < m.GetSize(); i++)
            out << m.vec[i] << endl;

      return out;
};
```