Федеральное государственное образовательное бюджетное учреждение
высшего профессионального образования
«Нижегородский Государственный Университет им.
Н.И.Лобачевского» (ННГУ)

Национальный исследовательский Университет
Институт Информационных Технологий Математики и Механики

Отчёт по лабораторной работе
Создание библиотеки с классами векторов и матриц

Выполнил:
студент группы 3821Б1ПМ3

Заботин М.А

Проверил:
заведующий лабораторией
суперкомпьютерных технологий и
высокопроизводительных вычислений

Лебедев И.Г

Нижний Новгород
2022 г.

Оглавление

1. Введение

Программирование - это интересный, полезный и увлекательный процесс, благодаря которому мы можем с помощью специальных команд обучать компьютер, делать для нас разнообразные полезные задачи, от выполнения операций с числами и навигации, до управления самолетами, спутниками и марсоходами.

Чтобы программировать сложные алгоритмы, необходимо постоянно пополнять свои знания о структурах и методах изучаемого языка программирования. Одной из таких сложных и интересных структур в языке C++ являются классы. Это по сути инструмент для создания новых типов переменных, наряду с int, float, bool и т.д. Классы используются, когда нам необходимо описать множество схожих объектов, например, животных в зоопарке, у каждого из которых есть вес, рост, количество особей. Но неэффективно описывать каждого объекта по отдельности, гораздо проще создать структуру, которая будет содержать в себе данные о каждом объекте в целом. Также классы облегчают работу с разными математическими объектами, так как могут содержать в себе различные функции (методы), присущие каждому из объектов, которые могут выполнить нужную задачу, а также набор разных значений, существующих у объекта (поля). Для объектов класса также можно выполнить перегрузку разных стандартных операций, по сути указать программе как нужно действовать с объектами класса, что значительно упростит работу с ними. Также, если не понятно, с каким типом данных внутри класса придется работать используют шаблоны, вместо того чтобы кодировать много одинаковых функций, различающихся только типом данных. Шаблоны позволяют в процессе работы использовать нужный тип данных в работе класса. В данной лабораторной работе также представлен еще один инструментарий, а именно вектор-итератор. На данном простом примере необходимо будет разобраться с принципом его работы и использованием при обходе структур данных. Для проверки используются Гугл-тесты. Они позволяют убедиться, что написанные классы функционируют правильно.

В данной лабораторной работе для изучения особенностей работы с классами, была поставлена задача: на языке «C++», используя шаблонные классы, а также перегрузки операций, необходимо написать программу, исходные файлы которой должны быть вынесены в статистическую библиотеку, для последующей удобной работы с алгебраическими векторами N-мерного пространства и матрицами.

2. Постановка задачи

Используя шаблонные классы и перегрузки операций, написать программу, которая позволяет удобно работать с векторами N-мерного пространства и квадратными матрицами, а именно, складывать, вычитать, умножать вектора и матрицы. Также дополнительным заданием было реализовать класс нижне-треугольных матриц и проверить правильность работы всех классов с помощью Гугл-тестов, написать которые необходимо самостоятельно. Также в задачу входит реализация класса вектора-итератора и освоение этого инструментария, так как это достаточно простой пример, чтобы понять, как он помогает при обходе более сложных структур данных, в которых уже не получится использовать обычные циклы. Исходные файлы, содержащие описание классов векторов и матриц должны быть вынесены в отдельную статистическую библиотеку, для последующей удобной работы с ними. Реализовать потоковые ввод и вывод для каждого класса, а также доступ к защищенным полям, а также провести практическую оценку времени работы программы на самых трудоемких алгоритмах, и сравнить её с теоретически предполагаемой.

Программа должна быть написана на языке «C++». Классы матриц должны быть наследниками класса вектора и по сути являться векторами векторов.

3. Руководство пользователя

Как таковой интерфейс работы с программой не реализован, так как задача была реализовать классы, для удобной работы с векторами и матрицами, но в исходном файле «main.cpp» написаны функции с тестами различных операций с векторами и квадратными и треугольными матрицами, а также тест вектора-итератора, которые выводят значения на экран, для проверки работоспособности различных методов класса. Если назначить авто-запускаемым этот файл, то можно будет убедиться в правильности работы классов. Функции закоментарены, и, чтобы испытать, как работает тот или иной метод классов, нужно убрать комментарии с нужной тестовой функции (См. Рис. 1)

```
int main()
{
  try
  {
    testtime();
    //testCreateMatix();
    //testMultMatrix();
    //testVectorIterator();
    //testMatrix();
    //testCreateTrangleMatrix();
    //testOperTrangleMatrix();
    //testMultTrangle();
  }
  catch (...)
  {
    cout << "exeption";
  }

  return 0;
}
```

Рисунок 1. Список тестовых функций.

Если назначить авто-запускаемым файл с Гугл-тестами, то запустится основной файл с ними (См. Рис. 2), который проведёт абсолютно все написанные тесты и выведет на экран результаты, были ли они пройдены. Выполняются абсолютно все тесты, для векторов, квадратных и треугольных матриц и будет понятно, в каких местах методы работают не так, как должны. На консоль выведется количество пройденных тестов, а также подробные их названия и время выполнения (См. Рис. 3)

```
#include <../gtest/gtest.h>

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Рисунок 2. Основной файл с тестами.

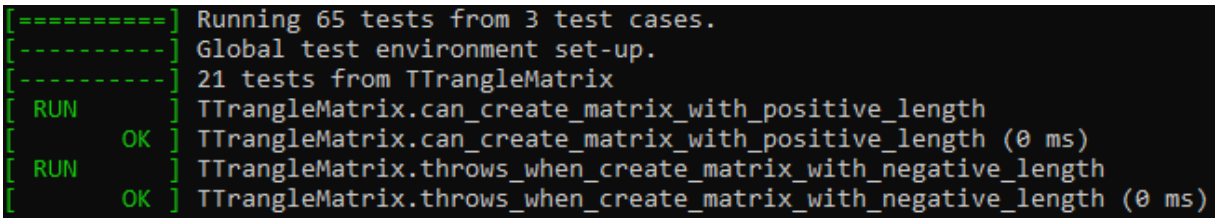

Рисунок 3. Вид консоли при выполнении Гугл-тестов.

Так как исходные файлы, содержащие описания классов вынесены в статистическую библиотеку «vectorLib.lib», написанный инструмент может быть использован в работе других программ разных людей, если понадобится работать с векторами или матрицами. Единственное что нужно в таком случае, это подключить данную библиотеку в своей программе и удобно работать с векторами и матрицами. В таком случае считаю необходимым описать методы и конструкторы классов векторов и матриц, чтобы дальнейшие пользователи знали, как работает тот или иной метод, а также перегрузка операции.

3.1 Класс векторов

Чтобы подключить библиотеку с вектором, необходимо в начале программы написать «`#include "Vector.h"`», после чего пользоваться инструментарием данного класса.

Создать вектор можно четырьмя способами, тип данных, который будет использоваться вектором указывается в треугольных скобках:

`TDynamicVector(size_t size = 1)`
Пример:
`TDynamicVector <int> V(10)`
	Конструктор, создающий вектор из size элементов, в каждой ячейке которого будет лежать 0. Размер по умолчанию равен 1.

`TDynamicVector(T* arr, size_t s)`
Пример:
`int*data[3]={1,3,4}`
`TDynamicVector<int> V(data,3)`
	Конструктор, создающий вектор, с помощью массива и его размера

`TDynamicVector(const TDynamicVector<T>& v)`
Пример:
`TDynamicVector<int> V(V1)`
	Конструктор копирования, принимающий на вход другой вектор и создающий его копию.

`TDynamicVector(TDynamicVector<T>&& v) noexcept` Пример здесь не приводится, так как компилятор сам решает, когда использовать конструктор перемещения.	Конструктор перемещения, принимающий на вход другой вектор и передающий всю информацию о нём новому вектору, удаляя исходный. Позволяет при больших объёмах данных не тратить ресурсы памяти на копирование.

Таблица 1.

Далее будет описание методов и перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`size_t size(void) const noexcept`	Возвращает длину вектора	size_t
`T& operator[](size_t ind)`	Перегрузка оператора доступа к ячейке вектора, возвращает ссылку на шаблонный тип и позволяет изменять его	T&
`const T& operator[](size_t ind) const`	Перегрузка оператора доступа к ячейке вектора, без возможности изменения.	const T&
`T& at(size_t ind)`	Метод класса, выдающий доступ к ячейке вектора с возможностью её изменения	T&
`const T& at(size_t ind) const`	Метод класса, выдающий доступ к ячейке вектора без возможности её изменения	const T&
`TDynamicVector<T>& operator=(const TDynamicVector<T>& v)`	Перегрузка оператора присваивания, которая переопределяет вектор и присваивает ему нужное значение	Ссылка на Вектор
`TDynamicVector<T>& operator=(TDynamicVector<T>&& v) noexcept`	Перегрузка оператора присваивания с перемещением, которая переопределяет вектор путем переопределения указателя на область памяти, а перемещаемые данные удаляются	Ссылка на Вектор
`TDynamicVector<T> operator+(const TDynamicVector<T>& v)`	Перегрузка оператора «+», при которой происходит покоординатное сложение векторов (если длины векторов разные то программа выдаст исключение)	Вектор
`TDynamicVector<T> operator-(const TDynamicVector<T>& v)`	Перегрузка оператора «-», при которой происходит покоординатное вычитание векторов (если длины векторов разные то программа выдаст исключение)	Вектор
`T operator*(const TDynamicVector<T>& v) noexcept(noexcept(T()))`	Перегрузка оператора «*», которая отвечает за вычисление скалярного произведения двух векторов (если длины векторов разные то программа выдаст исключение)	T
`TDynamicVector<T> operator+(T val)`	Перегрузка оператора «+», при которой на вход подается шаблонный элемент и он прибавляется к каждой координате вектора	Вектор
`TDynamicVector<T> operator-(T val)`	Перегрузка оператора «-», при которой на вход подается шаблонный элемент и он вычитается из каждой координаты вектора	Вектор

`TDynamicVector<T> operator*(T val)`	Перегрузка оператора «*», при которой на вход подается шаблонный элемент и он умножается на каждую координату вектора	Вектор
`bool operator==(const TDynamicVector<T>& v) const noexcept`	Оператор сравнивания (если длины векторов разные то возвращается ложь), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается истина, если нет, то ложь	bool
`bool operator!=(const TDynamicVector<T>& v) const noexcept`	Оператор сравнивания (если длины векторов разные то возвращается истина), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается ложь, если нет, то истина	bool
`friend ostream& operator<<(ostream& ostr, const TDynamicVector& v)`	Перегрузка оператора потокового вывода, красиво выводит вектор на консоль	Ссылку на объект ostream
`friend istream& operator>>(istream& istr, TDynamicVector& v)`	Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер вектора, после чего ввести каждую координату вектора	Ссылку на объект istream
`TVectorIterator<T> begin()`	Создаёт вектор-итератор начала вектора.	Вектор-итератор
`TVectorIterator<T> end()`	Создаёт вектор-итератор конца вектора.	Вектор-итератор

Таблица 2.

3.2 Класс квадратных матриц

Чтобы подключить библиотеку с матрицами, необходимо в начале программы написать «`#include "SquareMatrix.h"`», после чего пользоваться инструментарием данного класса.

Создать матрицу можно четырьмя способами, тип данных, который будет использоваться матрицей, указывается в треугольных скобках:

`TSquareMatrix(size_t size1)`
Пример:
`TSquareMatrix<int> M(5)`
	Конструктор, принимающий на вход количество строк и столбцов в матрице, а также заполняющий её нулями.

`TSquareMatrix(const TSquareMatrix<T>& mat) noexcept`
Пример:
`TSquareMatrix<int> M(M1)`
	Конструктор копирования, который на вход принимает другую матрицу и копирует данные в новую матрицу

`TSquareMatrix(TSquareMatrix<T>&& mat)`	Конструктор перемещения, который перемещает необходимые данные, удаляя

Примера как такового нет, он используется компилятора, когда невыгодно копирование больших объемов данных и проще просто перенаправить указатель	данные в исходном месте, чтобы не копировать.

`TSquareMatrix(size_t size,const T* arr)`
Пример:
`int*data[3]={1,3,4,5}`
`TSquareMatrix<int> M(2);`
	Конструктор, создающий матрицу необходимого размера и копирующий в неё данные из некоторого массива

Таблица 3.

Далее будет описание методов и перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`bool operator == (const TSquareMatrix<T>& mat) const noexcept`	Оператор сравнивания (если размеры матриц разные то возвращается ложь), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается истина, если нет, то ложь	bool
`bool operator != (const TSquareMatrix<T>& mat) const noexcept`	Оператор сравнивания (если размеры матриц разные то возвращается истина), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается ложь, если нет, то истина	bool
`TSquareMatrix<T> operator + (const TSquareMatrix<T>& mat)`	Перегрузка оператора «+», при которой выполняется поэлементное сложение	Матрица
`TSquareMatrix<T> operator - (const TSquareMatrix<T>& mat)`	Перегрузка оператора «-», при которой выполняется поэлементное вычитание	Матрица
`TSquareMatrix<T> operator * (const TSquareMatrix<T>& mat)`	Оператор умножения матриц, который вначале сравнивает размеры матриц и если они не совпадает, то выдаётся исключение. Если же размеры совпадают, то происходит умножение матриц строка на	Матрица

	столбец и создаётся новая результирующая матрица	
`TSquareMatrix<T> operator = (TSquareMatrix<T>&& mat)`	Перегрузка оператора присваивания с перемещением, которая переопределяет матрицу и присваивает ей нужное значение, удаляя данные из места, откуда производилось копирование	Матрица
`TSquareMatrix<T> operator = (const TSquareMatrix<T>& mat)`	Перегрузка оператора присваивания, которая переопределяет матрицу и присваивает ей нужное значение	Матрица
`ostream& operator <<(ostream& ostr, TSquareMatrix<T>& p)`	Перегрузка оператора потокового вывода, красиво выводит матрицу на консоль	Ссылка на объект ostream
`istream& operator >>(istream& istr, TSquareMatrix<T>& p)`	Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер матрицы, после чего ввести каждую ячейку матрицы, для того чтобы заполнить	Ссылка на объект istream

Таблица 4.

3.3 Класс треугольных матриц

Чтобы подключить библиотеку с нижне-треугольными матрицами, необходимо в начале программы написать «`#include "TrangleMatrix.h"`», после чего пользоваться инструментарием данного класса.

Создать матрицу можно четырьмя способами, тип данных, который будет использоваться матрицей, указывается в треугольных скобках:

`TTrangleMatrix(size_t size1)`
Пример:
`TTrangleMatrix<int> M(5)`
	Конструктор, принимающий на вход количество строк и столбцов в матрице, а также заполняющий её нулями.

`TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept`
Пример:
`TTrangleMatrix<int> M(M1)`
	Конструктор копирования, который на вход принимает другую матрицу и копирует данные в новую матрицу

`TTrangleMatrix(TTrangleMatrix<T>&& mat)` Примера как такового нет, он используется компилятора, когда невыгодно копирование больших объёмов данных и проще просто перенаправить указатель	Конструктор перемещения, который перемещает необходимые данные, удаляя данные в исходном месте, чтобы не копировать.

`TTrangleMatrix(size_t size, const T* arr)`
Пример:
`int*data[3]={1,3,4}`
`TTrangleMatrix<int> M(2);`
	Конструктор, создающий матрицу необходимого размера и копирующий в неё данные из некоторого массива

Таблица 5.

Далее будет описание методов и перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`bool operator == (const TTrangleMatrix<T>& mat) const noexcept`	Оператор сравнивания (если размеры матриц разные то возвращается ложь), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается истина, если нет, то ложь	bool
`bool operator != (const TTrangleMatrix<T>& mat) const noexcept`	Оператор сравнивания (если размеры матриц разные то возвращается истина), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается ложь, если нет, то истина	bool
`TTrangleMatrix<T> operator + (const TTrangleMatrix<T>& mat)`	Перегрузка оператора «+», при которой выполняется поэлементное сложение	Матрица
`TTrangleMatrix<T> operator - (const TTrangleMatrix<T>& mat)`	Перегрузка оператора «-», при которой выполняется поэлементное вычитание	Матрица
`TTrangleMatrix<T> operator * (const TTrangleMatrix<T>& mat)`	Оператор умножения матриц, который вначале сравнивает размеры матриц и если они не совпадает, то выдаётся исключение. Если же размеры совпадают, то происходит умножение матриц строка на столбец и создаётся новая результирующая матрица	Матрица
`TTrangleMatrix<T> operator = (TTrangleMatrix<T>&& mat)`	Перегрузка оператора присваивания с перемещением, которая переопределяет матрицу	Матрица

	и присваивает ей нужное значение, удаляя данные из места, откуда производилось копирование	
`TTrangleMatrix<T> operator = (const TTrangleMatrix<T>& mat)`	Перегрузка оператора присваивания, которая переопределяет матрицу и присваивает ей нужное значение	Матрица
`ostream& operator <<(ostream& ostr, TTrangleMatrix<T>& p)`	Перегрузка оператора потокового вывода, красиво выводит матрицу на консоль	Ссылка на объект ostream
`istream& operator >>(istream& istr, TTrangleMatrix<T>& p)`	Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер матрицы, после чего ввести каждую ячейку матрицы, для того чтобы заполнить	Ссылка на объект istream

Таблица 6.

3.4 Класс вектора-итератора

Чтобы подключить библиотеку с вектором-итератором, необходимо в начале программы написать «`#include "Vector.h"`», так как реализация данного класса находится вместе с реализацией обычного вектора, после чего пользоваться инструментарием данного класса.

Представлены 3 способа создания вектора-итератора, а именно:

`TVectorIterator(TDynamicVector<T>& v)`
Пример:
`TDynamicVector<int> V(10);`
`TVectorIterator<int> I(V);`
	Создание вектора-итератора с помощью обычного вектора, при чём индекс будет равен первой ячейке вектора

`TVectorIterator(TDynamicVector<T>& v, int ind)`
Пример:
`TDynamicVector<int> V(10);`
`TVectorIterator<int> I(V,3);`
	Создание вектора-итератора с помощью обычного вектора, и указания индекса, поэтому будет указание на конкретную ячейку вектора

`TVectorIterator(TVectorIterator<T>& iv)`
Пример:
`TDynamicVector<int> V(10);`
`TVectorIterator<int> I(V,3);`
`TVectorIterator<int> I1(I);`
	Конструктор копирования, который копирует данные из другого вектора-итератора, а именно вектор и индекс

Таблица 7.

Далее будет описание перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`bool operator ==(const TVectorIterator<T>& v)`	Перегрузка оператора сравнения (если равны индексы и вектора, то выдаётся истина, иначе ложь)	bool
`bool operator !=(const TVectorIterator<T>& v)`	Перегрузка оператора сравнения (если равны индексы и вектора, то выдаётся ложь, иначе истина)	bool
`TVectorIterator<T> operator++ ()`	Перегрузка оператора «++», отвечающая за увеличение индексации на 1, если индекс становится больше размера вектора, то он не увеличивается	Вектор-итератор
`TVectorIterator<T> operator-- ()`	Перегрузка оператора «--», отвечающая за уменьшение индексации на 1, если индекс становится равен нулю, то он не зануляется	Вектор-итератор
`T& operator*()`	Возвращает элемент из ячейки вектора по текущему индексу	T&

Таблица 8.

4. Руководство программиста

4.1 Описание структур данных

4.1.1 Классы

Класс векторов:

Этот шаблонный класс реализован в отдельном файле заголовка «Vector.h» и может быть использован в дальнейшем для удобной работы с векторами. (См. Рис.4)

```
template <typename T>
class TDynamicVector
{
protected:
  size_t sz;
  T* pMem;
public:
  TDynamicVector(size_t size = 1);
  TDynamicVector(T* arr, size_t s);
  TDynamicVector(const TDynamicVector<T>& v);
  TDynamicVector(TDynamicVector<T>&& v) noexcept;
  ~TDynamicVector();
  TDynamicVector<T>& operator=(const TDynamicVector<T>& v);
  TDynamicVector<T>& operator=(TDynamicVector<T>&& v) noexcept;

  size_t size(void) const noexcept;

  T& operator[](size_t ind);
  const T& operator[](size_t ind) const;

  T& at(size_t ind);
  const T& at(size_t ind) const;

  bool operator==(const TDynamicVector<T>& v) const noexcept;
  bool operator!=(const TDynamicVector<T>& v) const noexcept;

  TDynamicVector<T> operator+(T val);
  TDynamicVector<T> operator-(T val);
  TDynamicVector<T> operator*(T val);

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

Рисунок 4. Класс векторов.

Поля класса:

Поля класса находятся в режиме доступа «protected», что делает возможным дальнейшее использование их для наследников класса.

1) «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций
2) «sz», переменная типа «size_t», хранящая в себе данные о длине вектора, иными словами, размерность пространства

Методы класса:

Методы класса находятся в режиме доступа «public», что позволяет пользоваться ими при работе с объектами класса.

1) «TDynamicVector», это четыре конструктора, один из которых создаёт вектор заданной длины и заполняет его нулями (принимает на вход длину), совмещен с конструктором по умолчанию, создающим вектор единичной длины, конструктор копирования другого вектора и конструктор перемещения, очищающий исходную память (принимающий на вход ссылку на копируемый вектор), а также конструктор, создающий вектор с помощью массива и его длины.
2) «~TDynamicVector», деструктор, очищающий выделенную для массива память в конце работы программы.
3) «size()», метод, возвращающий размер вектора.
4) Перегрузка оператора «[]» реализована дважды, в случае если необходимо просто получить доступ к ячейке, либо получить доступ с возможностью изменения, принимают на вход номер ячейки.
5) Параллельно предыдущей перегрузки также есть 2 метода «at», которые также обеспечивают доступ к ячейке вектора.
6) Также дальше в описании класса идут перегрузки операций, а именно (*, +, -, =, ==, !=), принимающие на вход константные ссылки на вектора либо же шаблонный элемент.
7) Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.
8) Также есть методы «begin()» и «end()», отвечающие за обход вектора
9) В описании методов класса реализованы перегрузки операторов потокового ввода и вывода, принимающие на вход ссылку на объект класса векторов и поток, и возвращающие поток. (См. Рис.5)

```
friend istream& operator>>(istream& istr, TDynamicVector& v)
{
  for (size_t i = 0; i < v.sz; i++)
    istr >> v.pMem[i];
  return istr;
}
friend ostream& operator<<(ostream& ostr, const TDynamicVector& v)
{
  for (size_t i = 0; i < v.sz; i++)
    ostr << v.pMem[i] << ' ';
  return ostr;
}
```

Рисунок 5. Реализация перегрузок потокового ввода и вывода векторов.

Класс квадратных матриц:

Этот шаблонный класс реализован в отдельном файле заголовка «SquareMatrix.h», он является наследником класса векторов, и является по сути вектором векторов. Он позволяет удобно работать с квадратными матрицами.

(См. Рис.6)

```
template <class T>
class TSquareMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TSquareMatrix(size_t size1);
  TSquareMatrix(size_t size,const T* arr);
  TSquareMatrix(const TSquareMatrix<T>& mat) noexcept;
  TSquareMatrix(TSquareMatrix<T>&& mat);

  bool operator == (const TSquareMatrix<T>& mat) const noexcept;
  bool operator != (const TSquareMatrix<T>& mat) const noexcept;

  TSquareMatrix<T> operator + (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator - (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator * (const TSquareMatrix<T>& mat);

  TSquareMatrix<T> operator = (TSquareMatrix<T>&& mat);
  TSquareMatrix<T> operator = (const TSquareMatrix<T>& mat);
};
```

Рисунок 6. Класс квадратных матриц.

Поля класса:

У класса нет как таковых полей, вместо этого используются поля класса-предка, ведь матрица-это вектор, состоящий из векторов. Также, если учитывать, что рассматриваются квадратные матрицы, то можно не вводить вторую переменную для размера.

1) «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций
2) «sz», переменная типа «size_t», хранящая в себе данные о длине вектора, иными словами, размерность пространства

Методы класса:

Методы класса находятся в режиме доступа «public», что позволяет пользоваться ими при работе с объектами класса.

1) «TSquareMatrix», это 4 конструктора, один совмещённый, по умолчанию и инициализатор, принимающий на вход размер квадратной матрицы, два

других, это конструкторы копирования и перемещения, первый копирует данные, а другой передаёт память новому объекту, а старую уничтожает

2) Также дальше в описании класса идут перегрузки операций, а именно (*, +, -, =, ==, !=), принимающие на вход константные ссылки на вектора.
3) Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.
4) Вне описания методов класса реализованы перегрузки операторов потокового ввода и вывода, принимающие на вход ссылку на объект класса квадратных матриц и поток, и возвращающие поток. (См. Рис.7)
5) Деструктор отдельно не реализуется в матрицах, так как он неявно наследуется от класса вектора.

```
template<class T>
ostream& operator <<(ostream& ostr, TSquareMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j < p.size(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }

  return ostr;
}

template<class T>
istream& operator >>(istream& istr, TSquareMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TSquareMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;
}
```

Рисунок 7. Реализация перегрузок потокового ввода и вывода квадратных матриц.

Класс треугольных матриц:

Этот шаблонный класс реализован в отдельном файле заголовка «TrangleMatrix.h», он является наследником класса векторов, и является по сути вектором векторов. Он позволяет удобно работать с треугольными матрицами.

(См. Рис.8)

```
template <class T>
class TTrangleMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TTrangleMatrix(size_t size1);
  TTrangleMatrix(size_t size, const T* arr);
  TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept;
  TTrangleMatrix(TTrangleMatrix<T>&& mat);

  bool operator == (const TTrangleMatrix<T>& mat) const noexcept;
  bool operator != (const TTrangleMatrix<T>& mat) const noexcept;

  TTrangleMatrix<T> operator + (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator - (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator * (const TTrangleMatrix<T>& mat);

  TTrangleMatrix<T> operator = (TTrangleMatrix<T>&& mat);
  TTrangleMatrix<T> operator = (const TTrangleMatrix<T>& mat);
};
```

Рисунок 8. Класс треугольных матриц.

Поля класса:

У класса нет как таковых полей, вместо этого используются поля класса-предка, ведь матрица-это вектор, состоящий из векторов. Также, если учитывать, что рассматриваются треугольные матрицы, то можно не вводить вторую переменную для размера.

1) «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций
2) «sz», переменная типа «size_t», хранящая в себе данные о длине вектора, иными словами, размерность пространства

Методы класса:

Методы класса находятся в режиме доступа «public», что позволяет пользоваться ими при работе с объектами класса.

1) «TTrangleMatrix», это 4 конструктора, один совмещённый, по умолчанию и инициализатор, принимающий на вход размер квадратной матрицы, два

других, это конструкторы копирования и перемещения, первый копирует данные, а другой передает память новому объекту, а старую уничтожает

2) Также дальше в описании класса идут перегрузки операций, а именно (*, +, -, =, ==, !=), принимающие на вход константные ссылки на вектора.
3) Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.
4) Вне описания методов класса реализованы перегрузки операторов потокового ввода и вывода, принимающие на вход ссылку на объект класса квадратных матриц и поток, и возвращающие поток. (См. Рис.9)
5) Деструктор отдельно не реализуется в матрицах, так как он неявно наследуется от класса вектора.

```
template<class T>
ostream& operator <<(ostream& ostr, TTrangleMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j <= i; j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }
  return ostr;
}

template<class T>
istream& operator >>(istream& istr, TTrangleMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TTrangleMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;
}
```

Рисунок 9. Реализация перегрузок потокового ввода и вывода треугольных матриц.

Класс вектора-итератора:

Этот шаблонный класс реализован совместно с обычным вектором в файле заголовка «Vector.h». Данный класс упрощает обход по структурам данных, и рассматривается в данной лабораторной на достаточно простом примере с обходом вектора, чтобы освоить принцип его работы и применять в дальнейшем для более сложных структур данных (См. рис.10)

```
template <class T>
class TVectorIterator
{
public:
  TVectorIterator(TDynamicVector<T>& v);
  TVectorIterator(TDynamicVector<T>& v, int ind);
  TVectorIterator(TVectorIterator<T>& iv);

  bool operator ==(const TVectorIterator<T>& v);
  bool operator !=(const TVectorIterator<T>& v);

  TVectorIterator<T> operator++ ();
  TVectorIterator<T> operator-- ();

  T& operator*();
private:
  TDynamicVector<T>& vector;
  int index;
};
```

Рисунок 10. Объявление класса вектора-итератора.

Поля класса:

Полями данного класса являются ссылка на вектор «vector» типа «TDynamicVector<T>&», для того чтобы хранить данные вектора и удобно осуществлять по ним обход, а также, при надобности изменять их, а также индекс, по которому на данный момент находится вектор итератор «index» типа «int».

Методы класса:

1) Во-первых, методами класса являются три конструктора, а именно, по умолчанию, который принимает на вход обычный вектор и ставит индекс на 0, также принимающий вектор и индекс, на который необходимо поставить вектор-итератор, а также конструктор копирования, принимающий на вход другой вектор-итератор и копирующий данные.
2) Во-вторых, методами класса являются перегрузки операций сравнения (==, !=), чтобы мы могли сравнивать итераторы.

3) Также присутствуют (++, --), которые сдвигают индекс вперед или назад по вектору на единицу
4) Основной метод класса-это перегрузка оператора (*), которая по сути отвечает за извлечение данных вектора из ячейки по текущему индексу.

4.1.2 Библиотеки

В данной программе используется несколько библиотек:

1) «iostream», основная библиотека, включающая в себя элементы работы с потоковым вводом и выводом, а также содержащая базовые элементы.
2) «sstream», расширенная библиотека для работы с потоковыми данными, позволяющая создать переменную потокового типа, в которую записывать данные, чтобы проводить тесты.
3) Также классы вектора, вектора-итератора, квадратных и треугольных матриц реализованы в отдельных заголовочных файлах, «Vector.h», «SquareMatrix.h» и «TrangleMatrix.h» которые подключаются в другие файлы как библиотеки, чтобы работать с элементами созданных классов.
4) «time.h» и «stdlib.h» используются для подсчета времени работы алгоритма для эксперимента.
5) «../gtest/gtest.h», библиотека с гугл-тестами.

4.1.3 Функции

Про функции, являющиеся методами реализованных классов было рассказано ранее, а в данном абзаце речь пойдёт про тестовые функции в файле «main.cpp», которые проводят тесты разных методов классов и перегрузок операторов.

1) «testMultMatrix», тестовая функция, проверяющая правильность умножения матриц.
2) «testCreateMatix», функция, проверяющая правильность создания квадратной матрицы.
3) «testVectorIterator», функция, проверяющая работоспособность вектора-итератора
4) «testMatrix», проверяет правильность создания матрицы с помощью массива.
5) «testCreateTrangleMatrix», тест по созданию и выводу треугольной матрицы.
6) «testOperTrangleMatrix», проверяет правильность выполнения операций над треугольными матрицами.
7) «testMultTrangle», умножение треугольных матриц и проверка на правильность результата
8) «testtime», функция, позволяющая измерить время работы программы в миллисекундах

4.1.4 Типы данных

В данной программе используются как локальные переменные, существующие только внутри функций, предназначенные для каких-то локальных вычислений и работы, так и глобальные, которые распространяются на область программы. Но все переменные относятся к одному из типов:

1) «TDynamicVector<T>», образующая объект, вектор шаблонного типа «T».
2) «TSquareMatrix<T>», образующая объект, квадратную матрицу шаблонного типа «T».
3) «TTrangleMatrix<T>», образующая объект, треугольную матрицу шаблонного типа «T».
4) «TVectorIterator<T>», образующая объект, вектор-итератор шаблонного типа «T».
5) «int», создающий переменную целочисленного типа, которая может в дальнейшем понадобиться в расчётах.
6) «bool», логический тип данных, имеющий только 2 значения, истина или ложь.
7) «stringstream», универсальный потоковый тип данных, который можно использовать и для ввода, и для вывода.
8) «double», используемый для создания вещественного типа данных.
9) «T» шаблонный тип данных, который подразумевают под собой любой другой тип данных, и используемый для того, чтобы не описывать каждый метод класса сразу для всех типов данных, а описать один раз для шаблона, который сам везде поменяет тип.
10) «ostream», тип данных, для работы с потоковым выводом.
11) «istream», тип данных для работы с потоковым вводом.
12) «clock_t», тип, используемый для подсчёта времени работы алгоритма.
13) «size_t», переменная, используемая для объявления неотрицательных переменных.
14) Некоторые переменные могут быть заранее объявлены как «const», для того чтобы они были неизменными.
15) Также используются другие стандартные для языка «C++» типы данных, которые не нуждаются в отдельном описании.

4.2 Описание структуры программы

4.2.1 Описание структуры заголовочных файлов

Файл «Vector.h»:

Данный заголовочный файл содержит в себе реализацию класса вектора и вектора-итератора, вместе с реализацией их методов. Реализация методов происходит в сразу в заголовочном файле из-за особенностей компиляции шаблонных классов, при реализации которых в исходном файле могут возникнуть проблемы.

Сначала идет подключение необходимых для работы библиотек и использование стандартного пространства имен.

```
#pragma once
#include <iostream>
#include "VectorIterator.h"

using namespace std;
```

Фрагмент кода 1.

После чего идет объявление шаблонного класса вектора-итератора, без реализации, так как она будет представлена в дальнейшем, а также класса обычных векторов и заполнение его защищенных и общедоступных областей полями и методами,

конструкторами, деструктором, перегрузками операций, также прямо в объявление класса добавлены функции потокового ввода и вывода вектора.

```
template <class T>
class TVectorIterator;

template <typename T>
class TDynamicVector
{
protected:
  size_t sz;
  T* pMem;
public:
  TDynamicVector(size_t size = 1);
  TDynamicVector(T* arr, size_t s);
  TDynamicVector(const TDynamicVector<T>& v);
  TDynamicVector(TDynamicVector<T>&& v) noexcept;
  ~TDynamicVector();
  TDynamicVector<T>& operator=(const TDynamicVector<T>& v);
  TDynamicVector<T>& operator=(TDynamicVector<T>&& v) noexcept;

  size_t size(void) const noexcept;

  T& operator[](size_t ind);
  const T& operator[](size_t ind) const;

  T& at(size_t ind);
  const T& at(size_t ind) const;

  bool operator==(const TDynamicVector<T>& v) const noexcept;
  bool operator!=(const TDynamicVector<T>& v) const noexcept;

  TDynamicVector<T> operator+(T val);
  TDynamicVector<T> operator-(T val);
  TDynamicVector<T> operator*(T val);

  TDynamicVector<T> operator+(const TDynamicVector<T>& v);
  TDynamicVector<T> operator-(const TDynamicVector<T>& v);
  T operator*(const TDynamicVector<T>& v) noexcept(noexcept(T()));

  friend istream& operator>>(istream& istr, TDynamicVector& v)
  {
    for (size_t i = 0; i < v.sz; i++)
      istr >> v.pMem[i];
    return istr;
  }
  friend ostream& operator<<(ostream& ostr, const TDynamicVector& v)
  {
    for (size_t i = 0; i < v.sz; i++)
      ostr << v.pMem[i] << ' ';
    return ostr;
  }

  TVectorIterator<T> begin();
  TVectorIterator<T> end();

};
```

Фрагмент кода 2.

За этим начинается реализация методов класса, перед каждым методом при помощи комментариев будет указано, за что отвечает этот метод.

```
//Конструктор инициализатор, принимающий на вход длину вектора
template<typename T>
inline TDynamicVector<T>::TDynamicVector(size_t size)
```

```
{
  if (size <= 0)
    throw "Vector size <=0";
  sz = size;
  pMem = new T[sz];
  for (int i = 0; i < sz; i++)
    pMem[i] = {};
}

//Конструктор копирования, принимающий на вход ссылку на вектор
template<typename T>
inline TDynamicVector<T>::TDynamicVector(const TDynamicVector<T>& v)
{
  if (v.pMem == nullptr)
  {
    sz = 0;
    pMem = nullptr;
  }
  else
  {
    sz = v.sz;
    pMem = new T[sz];
    for (int i = 0; i < sz; i++)
      pMem[i] = v.pMem[i];
  }
}

//Конструктор перемещения, переопределяющий указатели и зануляющий исходную
память
template<typename T>
inline TDynamicVector<T>::TDynamicVector(TDynamicVector<T>&& v) noexcept
{
  sz = v.sz;
  pMem = v.pMem;

  v.sz = 0;
  v.pMem = nullptr;
}

//Конструктор, создающий вектор по длине и переданному массиву шаблонного типа
template<typename T>
inline TDynamicVector<T>::TDynamicVector(T* arr, size_t s)
{
  if (s <= 0)
    throw "Vector size <=0";
  if (arr == nullptr)
    throw "Vector arr empty";
  sz = s;
  pMem = new T[sz];
  for (int i = 0; i < sz; i++)
    pMem[i] = arr[i];
}

//Деструктор, очищающий выделенную память
template<typename T>
inline TDynamicVector<T>::~TDynamicVector()
{
  if (pMem != nullptr)
  {
    delete[] pMem;
    sz = 0;
    pMem = nullptr;
  }
}
```

```
//Метод, возвращающий длину вектора
template<typename T>
inline size_t TDynamicVector<T>::size(void) const noexcept
{
  return sz;
}

//Перегрузки оператора [], принимающие на вход номер ячейки вектора и отличающиеся тем, что на выходе мы получаем либо сам элемент, без возможности изменения, либо ячейку, содержимое которой можно поменять. Также эти перегрузки дублированы как функции
template<typename T>
inline T& TDynamicVector<T>::operator[](size_t ind)
{
  if (ind < sz)
    return pMem[ind];
  else
    throw out_of_range("negative index");
}

template<typename T>
inline const T& TDynamicVector<T>::operator[](size_t ind) const
{
  return pMem[ind];
}

template<typename T>
inline T& TDynamicVector<T>::at(size_t ind)
{
  if (pMem == nullptr)
    throw "emty memory";
  if (ind >= 0 && ind < sz)
    return pMem[ind];
  else
    throw "out of range";
}

template<typename T>
inline const T& TDynamicVector<T>::at(size_t ind) const
{
  if (pMem == nullptr)
    throw "emty memory";
  if (ind >= 0 && ind < sz)
    return pMem[ind];
  else
    throw "out of range";
}

//Перегрузка оператора копирования и копирования с перемещением, принимающие на вход ссылку на вектор
template<typename T>
inline TDynamicVector<T>& TDynamicVector<T>::operator=(const TDynamicVector<T>& v)
{
  if (this != &v)
  {
    if (pMem == nullptr)
      delete[] pMem;
    if (v.pMem == nullptr)
    {
      sz = 0;
      pMem = nullptr;
    }
    else
    {
```

```
      sz = v.sz;
      pMem = new T[sz];
      for (int i = 0; i < sz; i++)
        pMem[i] = v.pMem[i];
    }
  }
  else
    throw "copy itself";
  return *this;
}

template<typename T>
inline TDynamicVector<T>& TDynamicVector<T>::operator=(TDynamicVector<T>&& v)
noexcept
{
  if (this != &v)
  {
    if (pMem != nullptr)
      delete[] pMem;

    sz = v.sz;
    pMem = v.pMem;

    v.sz = 0;
    v.pMem = nullptr;
  }
  else
    throw "copy itself";
  return *this;
}

//Перегрузка оператора +, принимающая на вход ссылку на вектор
template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator+(const TDynamicVector<T>& v)
{
  if (sz != v.sz)
    throw "different sizes";
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] += v.pMem[i];
  return res;
}

//Перегрузка оператора -, принимающая на вход ссылку на вектор
template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator-(const TDynamicVector<T>& v)
{
  if (sz != v.sz)
    throw "different sizes";
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] -= v.pMem[i];
  return res;
}

//Перегрузка операторов +, -, *, принимающие на вход шаблонный скаляр
template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator+(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] += val;
  return res;
}
```

```
template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator-(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] -= val;
  return res;
}

template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator*(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] *= val;
  return res;
}

//Перегрузка операторов ==, != принимающие на вход ссылку на вектор
template<typename T>
inline bool TDynamicVector<T>::operator==(const TDynamicVector<T>& v) const
noexcept
{
  if (sz != v.sz)
    return false;
  for (int i = 0; i < sz; i++)
    if (pMem[i] != v.pMem[i])
      return false;
  return true;
}

template<typename T>
inline bool TDynamicVector<T>::operator!=(const TDynamicVector<T>& v) const
noexcept
{
  return !(this->operator==(v));
}

//Перегрузка оператора *, принимающая на вход ссылку на вектор
template<typename T>
inline T TDynamicVector<T>::operator*(const TDynamicVector<T>& v)
noexcept(noexcept(T()))
{
  if (sz != v.sz)
    throw "different sizes";
  T res = 0;
  for (int i = 0; i < sz; i++)
    res += pMem[i] * v.pMem[i];
  return res;
}

//Операторы начала и конца вектора, возвращающие векторы итераторы соответственно
с индексом начала или конца
template<typename T>
inline TVectorIterator<T> TDynamicVector<T>::begin()
{
  return TVectorIterator<T>(*this);
}

template<typename T>
inline TVectorIterator<T> TDynamicVector<T>::end()
{
  return TVectorIterator<T>(*this, this->size());
```

```
}

//Объявление шаблонного класса вектора-итератора
template <class T>
class TVectorIterator
{
public:
  TVectorIterator(TDynamicVector<T>& v);
  TVectorIterator(TDynamicVector<T>& v, int ind);
  TVectorIterator(TVectorIterator<T>& iv);

  bool operator ==(const TVectorIterator<T>& v);
  bool operator !=(const TVectorIterator<T>& v);

  TVectorIterator<T> operator++ ();
  TVectorIterator<T> operator-- ();

  T& operator*();
private:
  TDynamicVector<T>& vector;
  int index;
};

//Конструктор инициализатор, копирования и конструктор с указанием индекса
template<typename T>
inline TVectorIterator<T>::TVectorIterator(TDynamicVector<T>& v) :vector(v)
{
  index = 0;
}

template<typename T>
inline TVectorIterator<T>::TVectorIterator(TDynamicVector<T>& v, int ind)
:vector(v)
{
  index = ind;
}

template<typename T>
inline TVectorIterator<T>::TVectorIterator(TVectorIterator<T>& iv)
:vector(iv.vector), index(iv.index)
{
}

//Перегрузки Операторов сравнения для вектора-итератора
template<typename T>
inline bool TVectorIterator<T>::operator==(const TVectorIterator<T>& v)
{
  return (&vector == &(v.vector) && index == v.index);
}

template<typename T>
inline bool TVectorIterator<T>::operator!=(const TVectorIterator<T>& v)
{
  return !(this->operator==(v));
}

//Перегрузки операторов «++, --», в качестве сдвига текущего индекса
вектора-итератора вперёд или назад
template<typename T>
inline TVectorIterator<T> TVectorIterator<T>::operator++()
{
  TVectorIterator<T> res(*this);
  this->index++;
  res.index++;
```

```
  if (res.index > res.vector.size())
    res.index = res.vector.size();
  return res;
}

template<typename T>
inline TVectorIterator<T> TVectorIterator<T>::operator--()
{
  TVectorIterator<T> res(*this);
  res.index--;
  if (res.index < 0)
    res.index = 0;
  return res;
}

//Метод, возвращающий ссылку на элемент вектора по текущему индексу
template<typename T>
inline T& TVectorIterator<T>::operator*()
{
  return vector.at(index);
}
```

Фрагмент кода 3.

Файл «SqareMatrix.h»:

Данный заголовочный файл содержит в себе реализацию класса квадратных матриц, вместе с реализацией его методов. Реализация методов происходит в сразу в заголовочном файле из-за особенностей компиляции шаблонных классов, при реализации которых в исходном файле могут возникнуть проблемы.

Сначала идет подключение необходимых для работы библиотек. Так как в файле вектора уже подключены все необходимые библиотеки и пространство имён, нам всего лишь необходимо подключить файл заголовка вектора.

```
#pragma once
#include "Vector.h"
```

Фрагмент кода 4.

После чего идет объявление шаблонного класса матриц, который является наследником вектора и по сути является вектором, состоящим из векторов, и заполнение его защищенных и общедоступных областей полями и методами, конструкторами, перегрузками операций.

```
template <class T>
class TSquareMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TSquareMatrix(size_t size1);
  TSquareMatrix(size_t size,const T* arr);
  TSquareMatrix(const TSquareMatrix<T>& mat) noexcept;
  TSquareMatrix(TSquareMatrix<T>&& mat);

  bool operator == (const TSquareMatrix<T>& mat) const noexcept;
  bool operator != (const TSquareMatrix<T>& mat) const noexcept;

  TSquareMatrix<T> operator + (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator - (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator * (const TSquareMatrix<T>& mat);
```

```
  TSquareMatrix<T> operator = (TSquareMatrix<T>&& mat);
  TSquareMatrix<T> operator = (const TSquareMatrix<T>& mat);

};
```

Фрагмент кода 5.

После объявления класса начинается реализация методов класса, перед каждым методом при помощи комментариев будет указано, за что отвечает этот метод. Перегрузки операторов потокового ввода и вывода не являются методами класса, так как работают с потоком и являются внешними, поэтому для доступа к защищенным полям класса используются соответствующие методы класса.

```
//Конструктор, создающий матрицу с помощью массива
template<class T>
inline TSquareMatrix<T>::TSquareMatrix(size_t size, const T* arr) :
TSquareMatrix<T>(size)
{
  if (size == 0)
    throw out_of_range("size must be greater than zero");
  for (size_t i = 0; i < sz*sz; i++)
    pMem[i / sz][i % sz] = arr[i];
}

//Конструктор-инициализатор, принимающий на вход размер
template<class T>
inline TSquareMatrix<T>::TSquareMatrix(size_t size1):
TDynamicVector<TDynamicVector<T>> (size1)
{
  sz = size1;
  for (size_t i = 0; i < sz; i++)
    pMem[i] = TDynamicVector<T>(sz);
}

//Конструктор копирования
template<class T>
inline TSquareMatrix<T>::TSquareMatrix(const TSquareMatrix<T>& mat) noexcept :
TDynamicVector<TDynamicVector<T>> (mat)
{
  if (mat.pMem == nullptr)
  {
    pMem = nullptr;
    sz = 0;
  }
  else
  {
    sz = mat.sz;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < sz; i++)
      pMem[i] = mat.pMem[i];
  }
}

//Конструктор перемещения
template<class T>
inline TSquareMatrix<T>::TSquareMatrix(TSquareMatrix<T>&& mat)
{
  pMem = mat.pMem;
  sz = mat.sz;
  mat.pMem = nullptr;
  mat.sz = 0;
}

//Перегрузка оператора сравнения на равенство
template<class T>
```

```
inline bool TSquareMatrix<T>::operator==(const TSquareMatrix<T>& mat) const
noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator ==(mat);
}

//Перегрузка оператора сравнения на неравенство
template<class T>
inline bool TSquareMatrix<T>::operator!=(const TSquareMatrix<T>& mat) const
noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator !=(mat);
}

//Перегрузка оператора сложения матриц
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator+(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] + mat.pMem[i];
  return res;
}

//Перегрузка оператора вычитания матриц
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator-(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] - mat.pMem[i];
  return res;
}

//Перегрузка оператора умножения матриц
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator*(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t row = 0; row < sz; row++)
    for (size_t col = 0; col < sz; col++)
    {
      T sum = 0;
      for (size_t k = 0; k < sz; k++)
        sum += pMem[row][k] * mat.pMem[k][col];
      res.pMem[row][col] = sum;
    }
  return res;
}

//Перегрузка оператора присваивания с перемещением данных
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator=(TSquareMatrix<T>&& mat)
{
  if (this != &mat)
  {
    if (pMem != nullptr)
      delete[] pMem;
```

```
    pMem = mat.pMem;
    sz = mat.sz;
    mat.pMem = nullptr;
    mat.sz = 0;
  }
  else
    throw "copy itself";
  return *this;
}

//Перегрузка оператора присваивания
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator=(const TSquareMatrix<T>& mat)
{
  if (this != &mat)
  {
    sz = mat.sz;
    if (pMem != nullptr)
      delete[] pMem;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < mat.sz; i++)
      pMem[i] = mat.pMem[i];
  }
  else
    throw "copy itself";
  return *this;
}

//Перегрузка оператора потокового вывода
template<class T>
ostream& operator <<(ostream& ostr, TSquareMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j < p.size(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }

  return ostr;
}

//Перегрузка оператора потокового ввода
template<class T>
istream& operator >>(istream& istr, TSquareMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TSquareMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;

}
```

Фрагмент кода 6.

Файл «TrangleMatrix.h»:

Данный заголовочный файл содержит в себе реализацию класса треугольных матриц, вместе с реализацией его методов. Реализация методов происходит в сразу в заголовочном файле из-за особенностей компиляции шаблонных классов, при реализации которых в исходном файле могут возникнуть проблемы.

Сначала идет подключение необходимых для работы библиотек. Так как в файле вектора уже подключены все необходимые библиотеки и пространство имён, нам всего лишь необходимо подключить заголовочный файл вектора.

```
#pragma once
#include "Vector.h"
```

Фрагмент кода 7.

После чего идет объявление шаблонного класса треугольных матриц, который является наследником вектора и по сути является вектором, состоящим из векторов, и заполнение его защищенных и общедоступных областей полями и методами, конструкторами, перегрузками операций.

```
template <class T>
class TTrangleMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TTrangleMatrix(size_t size1);
  TTrangleMatrix(size_t size, const T* arr);
  TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept;
  TTrangleMatrix(TTrangleMatrix<T>&& mat);

  bool operator == (const TTrangleMatrix<T>& mat) const noexcept;
  bool operator != (const TTrangleMatrix<T>& mat) const noexcept;

  TTrangleMatrix<T> operator + (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator - (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator * (const TTrangleMatrix<T>& mat);

  TTrangleMatrix<T> operator = (TTrangleMatrix<T>&& mat);
  TTrangleMatrix<T> operator = (const TTrangleMatrix<T>& mat);

};
```

Фрагмент кода 8.

После объявления класса начинается реализация методов класса, перед каждым методом при помощи комментариев будет указано, за что отвечает этот метод. Перегрузки операторов потокового ввода и вывода не являются методами класса, так как работают с потоком и являются внешними, поэтому для доступа к защищенным полям класса используются соответствующие методы класса.

```
//Конструктор, создающий матрицу с помощью массива
template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(size_t size, const T* arr) :
TTrangleMatrix<T>(size)
{
  if (size == 0)
    throw out_of_range("size must be greater than zero");
  size_t iter = 0;
  for (size_t i = 0; i < sz; i++)
    for (int j = 0; j <= i; j++)
    {
```

```
      pMem[i][j] = arr[iter];
      iter++;
    }
}

//Конструктор-инициализатор, принимающий на вход размер
template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(size_t size1) :
TDynamicVector<TDynamicVector<T>>(size1)
{
  sz = size1;
  for (size_t i = 0; i < sz; i++)
    pMem[i] = TDynamicVector<T>(i+1);
}

//Конструктор копирования
template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept
{
  if (mat.pMem == nullptr)
  {
    pMem = nullptr;
    sz = 0;
  }
  else
  {
    sz = mat.sz;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < sz; i++)
      pMem[i] = mat.pMem[i];
  }
}

//Конструктор перемещения
template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(TTrangleMatrix<T>&& mat)
{
  pMem = mat.pMem;
  sz = mat.sz;
  mat.pMem = nullptr;
  mat.sz = 0;
}

//Перегрузка оператора сравнения на равенство
template<class T>
inline bool TSquareMatrix<T>::operator==(const TSquareMatrix<T>& mat) const
noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator ==(mat);
}

//Перегрузка оператора сравнения на неравенство
template<class T>
inline bool TTrangleMatrix<T>::operator!=(const TTrangleMatrix<T>& mat) const
noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator !=(mat);
}

//Перегрузка оператора сложения матриц
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator+(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
```

```
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] + mat.pMem[i];
  return res;
}

//Перегрузка оператора вычитания матриц
template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator-(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] - mat.pMem[i];
  return res;
}

//Перегрузка оператора умножения матриц
template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator*(const TTrangleMatrix<T>&
mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TTrangleMatrix<T> res(sz);
  for (size_t row = 0; row < sz; row++)
    for (size_t col = 0; col <= row; col++)
    {
      T sum = 0;
      for (size_t k = col; k <= row; k++)
        sum += pMem[row][k] * mat.pMem[k][col];
      res.pMem[row][col] = sum;
    }
  return res;
}

//Перегрузка оператора присваивания с перемещением данных
template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator=(TTrangleMatrix<T>&& mat)
{
  if (this != &mat)
  {
    if (pMem != nullptr)
      delete[] pMem;

    pMem = mat.pMem;
    sz = mat.sz;
    mat.pMem = nullptr;
    mat.sz = 0;
  }
  else
    throw "copy itself";
  return *this;
}

//Перегрузка оператора присваивания
template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator=(const TTrangleMatrix<T>&
mat)
{
  if (this != &mat)
  {
```

```
    sz = mat.sz;
    if (pMem != nullptr)
      delete[] pMem;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < mat.sz; i++)
      pMem[i] = mat.pMem[i];
  }
  else
    throw "copy itself";
  return *this;
}

//Перегрузка оператора потокового вывода
template<class T>
ostream& operator <<(ostream& ostr, TTrangleMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j <= i; j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }
  return ostr;
}

//Перегрузка оператора потокового ввода
template<class T>
istream& operator >>(istream& istr, TTrangleMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TTrangleMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;

}
```

Фрагмент кода 9.

4.2.2 Описание структуры исходных файлов

Файл «Vector.cpp»:

Так как все методы класса вектора и вектора-итератора реализованы в соответственном заголовочном файле, в исходном файле вектора нету никаких реализаций методов класса и перегрузок операторов. Данный исходный файл используется только для того, чтобы программа скомпилировалась, ведь заголовочные файлы сами по себе не компилируются, но если подключить заголовочный файл в исходном, то такая связка пройдет компиляцию успешно.

```
#include "Vector.h"
#include <iostream>
using namespace std;
```

Фрагмент кода 10.

Файл «Matrix.cpp»:

В этом случае ситуация такая же, как и с исходным файлом векторов, он нужен только для подключения файла заголовка и его компиляции.

```
#include <iostream>
#include "SquareMatrix.h"
#include "TrangleMatrix.h"
using namespace std;
```

Фрагмент кода 11.

Файл «main.cpp»:

Это основной исходный файл, в котором представлены тестовые функции, для проверки работы разных методов как класса векторов, так и матриц. Сначала идёт подключение всех необходимых библиотек, для работы программы, а также заголовочных файлов векторов и матриц, и использование стандартного пространства имен.

```
#include <iostream>
#include "Vector.h"
#include "SquareMatrix.h"
#include "TrangleMatrix.h"
#include <sstream>
using namespace std
```

Фрагмент кода 12.

Далее в файле идет реализация визуальных тестовых функций, при подключении которых на консоли выводится тест, позволяющий сравнить данные, полученные программой с теоретическими вычислениями, для убеждения в правильности работы алгоритмов и классов в целом.

```
//Тест времени работы
void testtime()
{
  clock_t t1, t2;
  int n = 100;
  for (int i = 100; i <= 500; i += 25)
  {
    TSquareMatrix<int> A(i);
    TSquareMatrix<int> B(i);

    t1 = clock();
    A * B;
    t2 = clock();

    cout << "Time:" << (t2 - t1) << " ms" << endl;
  }


}

//Проверка операций над квадратными матрицами
void testMatrix()
{
  constexpr int n = 3;
  TSquareMatrix<int> m1(n);
  TSquareMatrix<int> m2(m1);

  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    {
      m1[i][j] = i + j;
      m2[i][j] = i + j + 10;
    }
```

```
  }

  TSquareMatrix<int> m3(n);

  m3 = m1 + m2;

  cout << "m1=\n" << m1;
  cout << "m2=\n" << m2;
  cout << "m3=m1+m2\n" << m3;
}

//Проверка умножения матриц
void testMultMatrix()
{
  constexpr int n = 2;
  TSquareMatrix<int> m1(n);
  TSquareMatrix<int> m2(n);
  TSquareMatrix<int> m3(n);

  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    {
      m1[i][j] = i + j;
      m2[i][j] = i + j + 1;
    }
  }

  m2[0][1] = 3;

  cout << "m1=" << m1 << endl;
  cout << "m2=" << m2 << endl;

  m3 = m1 * m2;
  cout << m3;
}

//Проверка создания матрицы с помощью массива
void testCreateMatix()
{
  const int arr[4] = {3, 2, 4, 5};
  TSquareMatrix<int> tmp(2, arr);
  cout << tmp;
}

//Проверка работоспособности вектора итератора
void testVectorIterator()
{
  constexpr int n = 3;
  TDynamicVector<int> a(n);

  for (int i = 0; i < n; i++)
    a[i] = i + 1;

  TDynamicVector<int> b = a;
  TDynamicVector<int> c;

  c = a + b;
  cout << c;


  //for (auto i = c.begin(); i != c.end(); ++i)
  //  cout << *i;
```

```
  for (auto& v : c)
  {
    v = v + 1;
  }
  cout << c;

}

//Проверка правильности создания треугольной матрицы
void testCreateTrangleMatrix()
{
  TTrangleMatrix<int> m(3);
  m[0][0] = 1;
  m[1][0] = 2;
  m[1][1] = 3;
  m[2][0] = 4;
  m[2][1] = 5;
  m[2][2] = 6;
  cout << m;
}

//Проверка операций с треугольными матрицами
void testOperTrangleMatrix()
{
  int k = 1;
  TTrangleMatrix<int> m(3);
  TTrangleMatrix<int> m1(m);
  for (int i = 0; i < m.size(); i++)
    for (int j = 0; j <= i; j++)
    {
      m[i][j] = k;
      m1[i][j] = k + 1;
      k++;
    }
  TTrangleMatrix<int> m2(3);
  m2 = m + m1;
  cout << "m1=\n" << m;
  cout << "m2=\n" << m1;
  cout << "m3=m1+m2\n" << m2;
}

//Проверка умножения треугольных матриц
void testMultTrangle()
{
  int arr[3] = { 1,2,3 };
  int arr1[3] = { 2,3,1 };

  TTrangleMatrix<int> m(2, arr);
  TTrangleMatrix<int> m1(2, arr1);
  TTrangleMatrix<int> m2(2);
  m2 = m * m1;
  cout << "m1=\n" << m;
  cout << "m2=\n" << m1;
  cout << "m3=m1*m2\n" << m2;
}
```

Фрагмент кода 13.

После этого начинается основная функция «main», в которой в блоке «try»-«catch», вызываются тестовые функции и если возникает исключение, то оно обрабатывается и выводится на консоль.

```
int main()
{
  try
```

```
  {
    //testCreateMatix();
    //testMultMatrix();
    //testVectorIterator();
    //testMatrix();
    //testCreateTrangleMatrix();
    //testOperTrangleMatrix();
    testMultTrangle();
  }
  catch (exception&ex)
  {
    cout << ex.what();
  }
  return 0;

}
```

Фрагмент кода 14.

Файлы с Гугл-тестами

Гугл тесты используются для определения правильности работы написанных алгоритмов, но не являются визуальными, как тесты в основном исходном файле. При запуске Гугл-тестов на консоль выводятся лишь уведомления о запуске теста, а также его окончании, статусе теста (пройден ли он) и время в миллисекундах, которое было затрачено на выполнение тестов. Так происходит для всех тестов и отличие тут в том, что надо быть уверенным, что машина в принципе считает правильные в теории результаты, ведь составлять тесты надо было самостоятельно. Всего есть три придаточных файла с тестами, которые содержат тесты одного из написанных классов, а также один главный, который отвечает за запуск всех остальных тестов. Вот что содержится в основном файле Гугл-тестов:

```
#include <../gtest/gtest.h>
int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();

}
```

Фрагмент кода 15.

В остальных файлах содержатся лишь тесты, выполняющие простые проверки для выявления ошибок. Пример содержимого придаточных исходных файлов:

```
#include "SquareMatrix.h"
#include <../gtest/gtest.h>
TEST(TSquareMatrix, can_create_matrix_with_positive_length)
{
  ASSERT_NO_THROW(TSquareMatrix<int> m(5));

}

TEST(TSquareMatrix, copied_matrix_is_equal_to_source_one)
{
  TSquareMatrix<int> m(5);
  TSquareMatrix<int> m1(m);
  EXPECT_EQ(m, m1);

}
```

Фрагмент кода 16.

4.3 Описание алгоритмов

Большинство алгоритмов данной программы работают достаточно просто, взять, например, конструкторы, которые выделяют память при создании переменных и заполняют ячейки вектора или матрицы соответствующими значениями, которые необходимы пользователю. Или деструктор, который очищает выделенную память, чтобы избежать переполнения памяти. Алгоритм доступа к защищенным полям очень примитивен и просто возвращает значение защищенного поля. Также практически все алгоритмы при работе с векторами являются простыми и интуитивно понятными, например, умножение на скаляр просто умножает каждую ячейку вектора, сложение и вычитание векторов возвращает новый вектор с координатами, являющимися суммой координат, присваивание выделяет для вектора новую память и заполняет значениями, которые необходимо было присвоить. Либо потоковые операции, которые красиво распечатывают вектор на консоли, либо наоборот считывают длину и данные для вектора и заполняют его.

Больше всего интересны алгоритмы, работающие с матрицами. Конструкторы и деструкторы конечно, также просты в реализации и просто выделяют память для вектора векторов или очищают ее. Оператор сравнения вынесены через представление матрицы, как вектора векторов и вызовом функций перегрузки векторов, что является полиморфизмом. Операторы присваивания и перемещения во многом повторяют соответствующие конструкторы. Наибольший интерес вызывают алгоритмы суммирования и перемножения матриц между собой. Они основаны на математических правилах сложения и умножения матриц. И если сложение матриц не совсем интересный алгоритм, ведь по сути значения в ячейках с одинаковыми индексами складываются и переносятся в соответствующую ячейку новой результирующей матрицы (но сначала проверяются размерности, ведь нельзя складывать матрицы, если количества столбцов и строк не совпадают), то умножение матрицы очень интересный алгоритм. Сложность для сложения двух матриц n*n оценивается, как $O(n^2)$.

Умножение матриц на практике выполняется по формуле $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, и является более трудоемким процессом. Главное при умножении матриц, чтобы оно получилось, необходимо, чтобы количество столбцов первой матрицы совпадало с количеством строк второй матрицы, так как матрицы умножаются строка на столбец, каждые ячейки этих столбцов перемножаются и складываются в общую сумму, которую потом записывают в ячейку результирующей матрицы, а сама ячейка определяется какими по счёту строка и столбец умножаются, Но в данном случае реализованы квадратные и нижние-треугольные матрицы и количество строк определяет количество столбцов, поэтому сравнивается один параметр. Если исходные матрицы были размеров n*n и n*n, то результирующая будет размером n*n. Для нахождения всех элементов которой понадобится n*n раз вычислений суммы перемножения строки на столбец. Алгоритм исполняется при помощи трех вложенных циклов и для квадратной матрицы n*n сложность выполнения оценивается теоретически как $O(n^3)$.

Сейчас будет продемонстрирован самый интересный и трудоемкий из всех алгоритмов в данной лабораторной работе, а именно алгоритм умножения квадратных матриц:

1) В начале идет перегрузка операции умножения для матриц, возвращает данная перегрузка матрицу, а на вход принимает ссылку на матрицу.

```
template<class T>
```

```
inline TSquareMatrix<T> TSquareMatrix<T>::operator*(const TSquareMatrix<T>&
mat)
{
```

Фрагмент кода 17.

2) После чего идет проверка на то, что количество столбцов первой матрицы равно количеству строк второй матрицы, если условие выполнено, то алгоритм продолжает работу, если же нет, то прекращает работу, выдавая исключение с указанием на проблему: несовместимые размеры матриц для умножения.

```
if (sz != mat.sz)
  throw "different sizes";
```

Фрагмент кода 18.

3) Если размеры совпали, то сначала создаётся локальная результирующая матрица, которая впоследствии будет возвращена, как результат работы умножения, потом запускаются 2 вложенных цикла, которые переберут все ячейки новой матрицы и на каждом шаге будут сначала обнулять временную переменную, потом высчитывать новое ее значение, как сумму произведений координат соответствующей строки и столбца матриц, а также присваивать значение этой переменной необходимому элементу новой матрицы. После чего функция возвращает результирующую матрицу.

```
TSquareMatrix<T> res(sz);
  for (size_t row = 0; row < sz; row++)
   for (size_t col = 0; col < sz; col++)
   {
     T sum = 0;
     for (size_t k = 0; k < sz; k++)
       sum += pMem[row][k] * mat.pMem[k][col];
     res.pMem[row][col] = sum;
   }
return res;
```

Фрагмент кода 19.

В этом и состоит самый интересный и трудоемкий алгоритм, представленный в данной лабораторной работе (См. Рис.11).

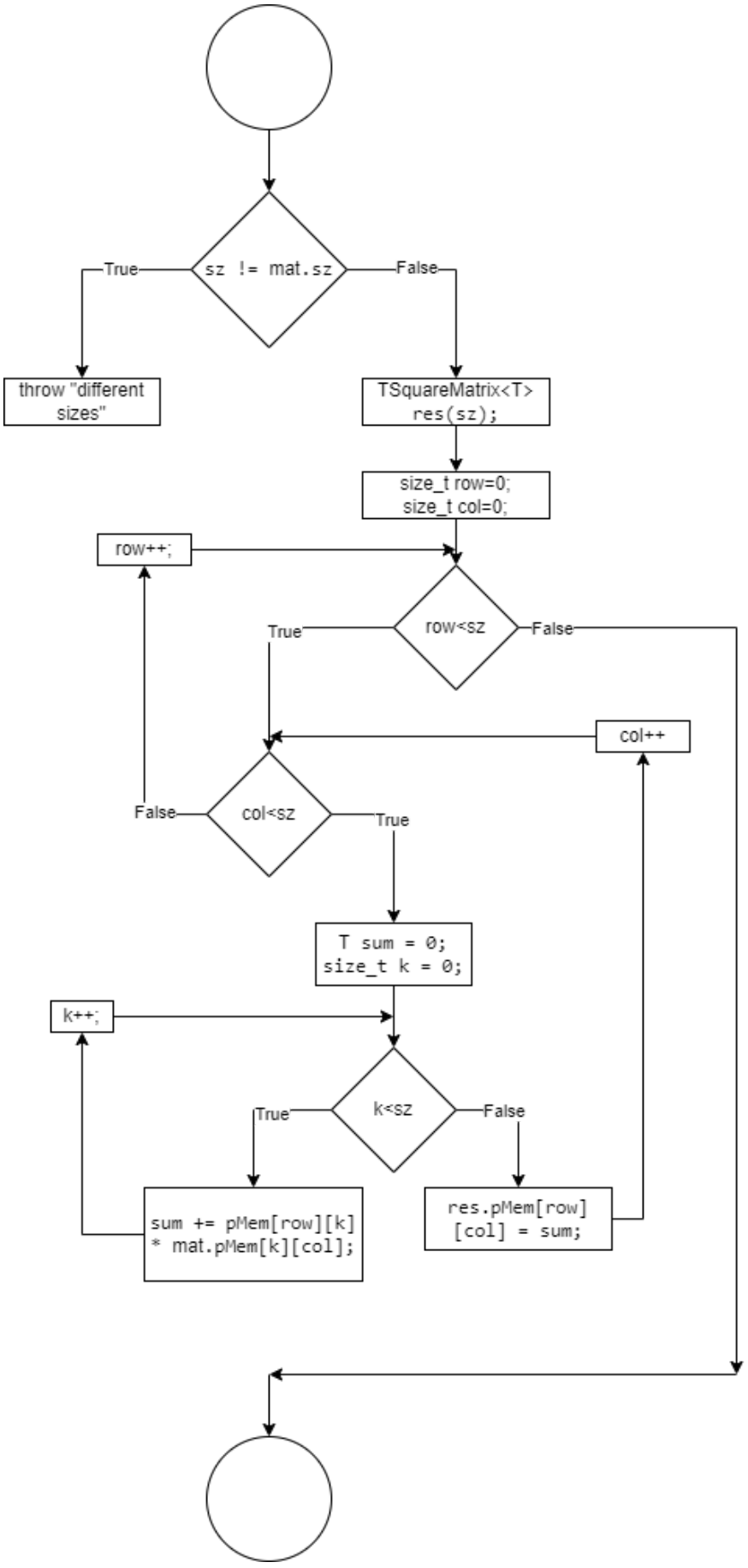

Рисунок 11. Блок-схема алгоритма умножения матриц.

5. Эксперименты

Так как самыми затратными по времени исполнения алгоритмами являются матричные сложение и умножение, именно их время работы нужно сравнивать с теоретически предполагаемым. Чтобы измерить время работы программы потребуется соответствующая функция «testtime», которую сначала адаптируем под сложение, а потом под умножение. Также нам потребуется конструктор матрицы, который заполняет каждую ячейку фиксированным значением.

```
void testtime()
{
  clock_t t1, t2;
  int n = 1000;
  for (int i = 1000; i <= 5000; i += 250)
  {
    TSquareMatrix<int> A(i);
    TSquareMatrix<int> B(i);

    t1 = clock();
    A + B;
    t2 = clock();
    cout << "Time:" << (t2 - t1) << " ms" << endl;
  }

}
```

Фрагмент кода 20.

Сначала запустим этот тест размера 1000 и посмотрим, сколько же времени затратит данный алгоритм. (См. Рис.12)

```
Time:9 ms
```

Рисунок 12. Время работы алгоритма сложения, при матрице 1000*1000.

Сложение матриц оценивается как $O\left(n^2\right)$, так же, как и выделение памяти для создания временной матрицы. Теоретически время будет равно $T(n) \approx an^2$. Подставив наши данные, получаем, что $a \approx 9,07 \bullet 10^{-6}$. После этого можно провести несколько экспериментов и составить по ним таблицу с двумя графиками, практических и теоретических значений, после чего сравнить результаты.

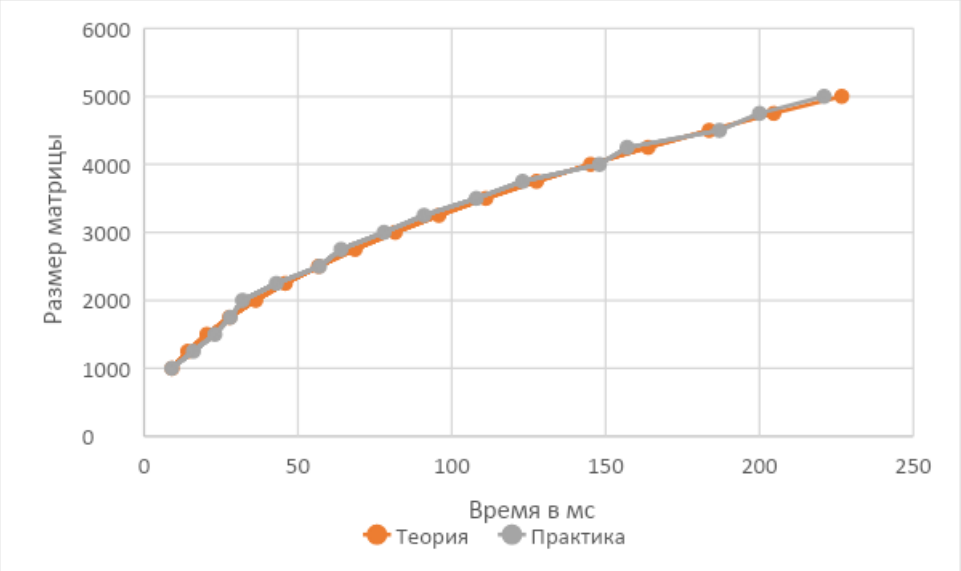

График 1.

Получив практические данные по времени для каждой размерности матриц, мы получили график и теперь можем сравнить. Можно видеть, что практические данные не сильно отличаются от теоретически предполагаемых, есть конечно несколько отклонений, но в целом все данные сходятся.

Далее проверим алгоритм умножения матриц, для этого всего лишь необходимо операцию сложения заменить на умножение. На этот раз диапазон размеров матрицы будет меньше и начинаться от 100. Сам алгоритм умножения работает за $O(n^3)$ однако, помимо умножения в алгоритме происходит выделение памяти для n^2 элементов, требующее bn^2 времени. Итого получим время работы $T(n) \approx an^3 + bn^2$. Коэффициенты a и b несложно найти из имеющихся данных ($a \approx 6,75 \cdot 10^{-6}$, $b \approx -7,24 \cdot 10^{-5}$). После этого необходимо провести серию экспериментов, также, как и в случае со сложением и занести данные в таблицу, после чего построить график и анализировать данные.

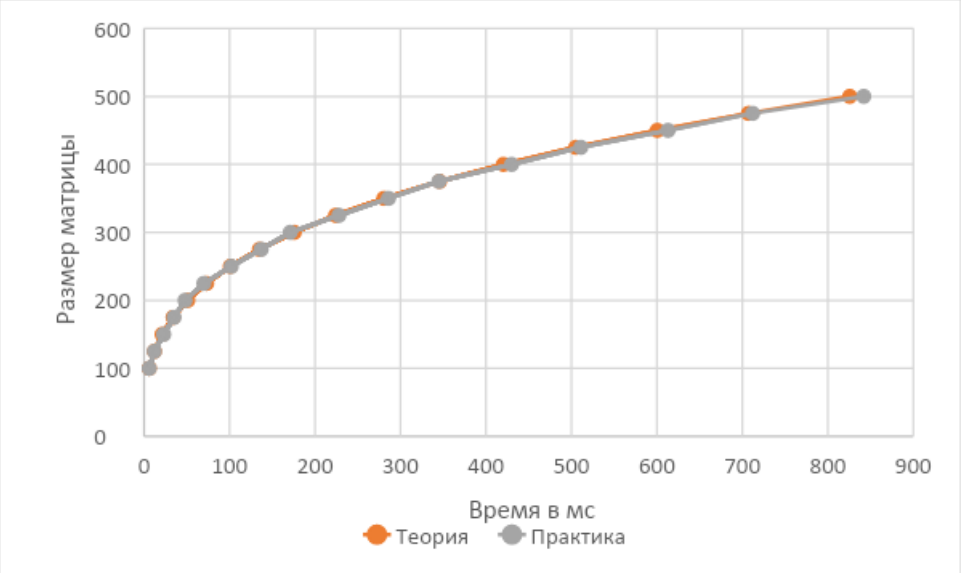

График 2.

Проанализировав данные, можно увидеть, что также, как и в случае со сложением матриц, практические данные не сильно отличаются от теоретически рассчитанных.

6. Заключение

В результате проведенной лабораторной работы, на языке программирования «C++» была написана программа, содержащая в себе описание и реализацию четырех шаблонных классов, вектора, вектора-итератора квадратных матриц, треугольных матриц, причем классы матриц являются наследниками вектора и по сути матрицы представляют собой вектора, состоящие из векторов. Эти классы полностью описывают весь необходимый для работы с векторами и матрицами функционал и позволяют упростить работу с ними в дальнейшем. Также, исходные и заголовочные файлы с классами векторов и матриц выведены в статистическую библиотеку, что упрощает дальнейшее использование этого полезного инструментария как мной, так и другими пользователями, всё что остаётся сделать, это подключать в дальнейших программах эту библиотеку при необходимости и пользоваться функциями, описанными для векторов или матриц.

Также было проведено исследование времени работы программы на самых трудоемких алгоритмах, а именно сложение и умножение матриц. Проведено сравнение теоретически предполагаемого времени работы этих алгоритмов, с полученными на практике для разных размеров матриц. В результате исследования было получено, что практическое время выполнения алгоритмов не сильно отличается от теоретически предположенного времени и практически с ним совпадает.

Шаблонные классы являются полезным и интересным объектом языка программирования «C++». Они позволяют создавать классы, то есть по сути новые переменные, со своими данными и функциями, не несколько раз под разные типы данных, а один раз. Дальнейшую работу выполнят шаблоны, которые поставят те типы данных, которые будут необходимы пользователю. Классы очень полезны, когда мы хотим создать некоторую структуру (как например в данной лабораторной работе вектора и матрицы), у которой есть сложные данные, и функции, необходимые для работы с этой структурой. Одними из таких функций являются перегрузки операторов, которые по сути дают понимание компилятору, как поступать при выполнении стандартных операций над элементами данных классов. В ходе работы были полностью изучены методы работы с шаблонными классами, перегрузками операторов и реализованы тесты на практике, а также были написаны и успешно пройдены Гугл-тесты, позволяющие удостовериться, что все методы классов работают правильно. На данном достаточно простом для понимания примере был освоен вектор-итератор, который используется для обхода структур данных, позволяет напрямую обращаться к элементу структуры данных. Если сейчас вектор можно обойти с помощью цикла, то в дальнейшем для более сложных типов данных вектор-итератор сможет помочь, когда обычный цикл не справится. К тому же было изучено создание статистической библиотеки с исходными и заголовочными файлами, содержащими реализацию классов векторов и матриц, упрощающее использование этого инструментария в дальнейшем как мной, так и другими людьми. В дальнейшем необходимо продолжать совершенствовать свои знания о языке программирования «C++», изучать более сложные и интересные алгоритмы и структуры, а также писать более новые, совершенные и полезные программы.

7. Литература

1. Т.А. Павловская Учебник по программированию на языках высокого уровня(C/C++) – Режим доступа: http://cph.phys.spbu.ru/documents/First/books/7.pdf

2. Бьерн Страуструп. Язык программирования C++ - Режим доступа: http://8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf

8. Приложения

8.1 Реализация класса вектора и вектора-итератора

```
#pragma once
#include <iostream>
#include "VectorIterator.h"
using namespace std;

template <class T>
class TVectorIterator;

template <typename T>
class TDynamicVector
{
protected:
  size_t sz;
  T* pMem;
public:
  TDynamicVector(size_t size = 1);
  TDynamicVector(T* arr, size_t s);
  TDynamicVector(const TDynamicVector<T>& v);
  TDynamicVector(TDynamicVector<T>&& v) noexcept;
  ~TDynamicVector();
  TDynamicVector<T>& operator=(const TDynamicVector<T>& v);
  TDynamicVector<T>& operator=(TDynamicVector<T>&& v) noexcept;

  size_t size(void) const noexcept;

  T& operator[](size_t ind);
  const T& operator[](size_t ind) const;

  T& at(size_t ind);
  const T& at(size_t ind) const;

  bool operator==(const TDynamicVector<T>& v) const noexcept;
  bool operator!=(const TDynamicVector<T>& v) const noexcept;

  TDynamicVector<T> operator+(T val);
  TDynamicVector<T> operator-(T val);
  TDynamicVector<T> operator*(T val);

  TDynamicVector<T> operator+(const TDynamicVector<T>& v);
  TDynamicVector<T> operator-(const TDynamicVector<T>& v);
  T operator*(const TDynamicVector<T>& v) noexcept(noexcept(T()));

  friend istream& operator>>(istream& istr, TDynamicVector& v)
  {
    for (size_t i = 0; i < v.sz; i++)
      istr >> v.pMem[i];
    return istr;
  }
  friend ostream& operator<<(ostream& ostr, const TDynamicVector& v)
  {
    for (size_t i = 0; i < v.sz; i++)
      ostr << v.pMem[i] << ' ';
    return ostr;
  }

  TVectorIterator<T> begin();
  TVectorIterator<T> end();
};

template<typename T>
```

```
inline TDynamicVector<T>::TDynamicVector(size_t size)
{
  if (size <= 0)
    throw "Vector size <=0";
  sz = size;
  pMem = new T[sz];
  for (int i = 0; i < sz; i++)
    pMem[i] = {};
}

template<typename T>
inline TDynamicVector<T>::TDynamicVector(T* arr, size_t s)
{
  if (s <= 0)
    throw "Vector size <=0";
  if (arr == nullptr)
    throw "Vector arr empty";
  sz = s;
  pMem = new T[sz];
  for (int i = 0; i < sz; i++)
    pMem[i] = arr[i];
}

template<typename T>
inline TDynamicVector<T>::TDynamicVector(const TDynamicVector<T>& v)
{
  if (v.pMem == nullptr)
  {
    sz = 0;
    pMem = nullptr;
  }
  else
  {
    sz = v.sz;
    pMem = new T[sz];
    for (int i = 0; i < sz; i++)
      pMem[i] = v.pMem[i];
  }
}

template<typename T>
inline TDynamicVector<T>::TDynamicVector(TDynamicVector<T>&& v) noexcept
{
  sz = v.sz;
  pMem = v.pMem;

  v.sz = 0;
  v.pMem = nullptr;
}

template<typename T>
inline TDynamicVector<T>::~TDynamicVector()
{
  if (pMem != nullptr)
  {
    delete[] pMem;
    sz = 0;
    pMem = nullptr;
  }
}

template<typename T>
inline TDynamicVector<T>& TDynamicVector<T>::operator=(const TDynamicVector<T>& v)
{
```

```
  if (this != &v)
  {
    if (pMem == nullptr)
      delete[] pMem;
    if (v.pMem == nullptr)
    {
      sz = 0;
      pMem = nullptr;
    }
    else
    {
      sz = v.sz;
      pMem = new T[sz];
      for (int i = 0; i < sz; i++)
        pMem[i] = v.pMem[i];
    }
  }
  else
    throw "copy itself";
  return *this;
}

template<typename T>
inline TDynamicVector<T>& TDynamicVector<T>::operator=(TDynamicVector<T>&& v) noexcept
{
  if (this != &v)
  {
    if (pMem != nullptr)
      delete[] pMem;

    sz = v.sz;
    pMem = v.pMem;

    v.sz = 0;
    v.pMem = nullptr;
  }
  else
    throw "copy itself";
  return *this;
}

template<typename T>
inline size_t TDynamicVector<T>::size(void) const noexcept
{
  return sz;
}

template<typename T>
inline T& TDynamicVector<T>::operator[](size_t ind)
{
  if (ind < sz)
    return pMem[ind];
  else
    throw out_of_range("negative index");
}

template<typename T>
inline const T& TDynamicVector<T>::operator[](size_t ind) const
{
  return pMem[ind];
}

template<typename T>
inline T& TDynamicVector<T>::at(size_t ind)
```

```
{
  if (pMem == nullptr)
    throw "emty memory";
  if (ind >= 0 && ind < sz)
    return pMem[ind];
  else
    throw "out of range";
}

template<typename T>
inline const T& TDynamicVector<T>::at(size_t ind) const
{
  if (pMem == nullptr)
    throw "emty memory";
  if (ind >= 0 && ind < sz)
    return pMem[ind];
  else
    throw "out of range";
}

template<typename T>
inline bool TDynamicVector<T>::operator==(const TDynamicVector<T>& v) const noexcept
{
  if (sz != v.sz)
    return false;
  for (int i = 0; i < sz; i++)
    if (pMem[i] != v.pMem[i])
      return false;
  return true;
}

template<typename T>
inline bool TDynamicVector<T>::operator!=(const TDynamicVector<T>& v) const noexcept
{
  return !(this->operator==(v));
}

template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator+(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] += val;
  return res;
}

template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator-(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] -= val;
  return res;
}

template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator*(T val)
{
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] *= val;
  return res;
}
```

```
template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator+(const TDynamicVector<T>& v)
{
  if (sz != v.sz)
    throw "different sizes";
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] += v.pMem[i];
  return res;
}

template<typename T>
inline TDynamicVector<T> TDynamicVector<T>::operator-(const TDynamicVector<T>& v)
{
  if (sz != v.sz)
    throw "different sizes";
  TDynamicVector<T> res(*this);
  for (int i = 0; i < sz; i++)
    res.pMem[i] -= v.pMem[i];
  return res;
}

template<typename T>
inline T TDynamicVector<T>::operator*(const TDynamicVector<T>& v) noexcept(noexcept(T()))
{
  if (sz != v.sz)
    throw "different sizes";
  T res = 0;
  for (int i = 0; i < sz; i++)
    res += pMem[i] * v.pMem[i];
  return res;
}

template<typename T>
inline TVectorIterator<T> TDynamicVector<T>::begin()
{
  return TVectorIterator<T>(*this);
}

template<typename T>
inline TVectorIterator<T> TDynamicVector<T>::end()
{
  return TVectorIterator<T>(*this, this->size());
}

////////////////////////////////

template <class T>
class TDynamicVectorIterator
{
public:
  TDynamicVectorIterator(TDynamicVector<T>& v);
  TDynamicVectorIterator(TDynamicVector<T>& v, int ind);
  TDynamicVectorIterator(TDynamicVectorIterator<T>& iv);

  bool operator ==(const TDynamicVectorIterator<T>& v);
  bool operator !=(const TDynamicVectorIterator<T>& v);

  TDynamicVectorIterator<T> operator++ ();
  TDynamicVectorIterator<T> operator-- ();

  T& operator*();
private:
  TDynamicVector<T>& vector;
```

```
  int index;
};

template<typename T>
inline TDynamicVectorIterator<T>::TDynamicVectorIterator(TDynamicVector<T>& v) :vector(v)
{
  index = 0;
}

template<typename T>
inline TDynamicVectorIterator<T>::TDynamicVectorIterator(TDynamicVector<T>& v, int ind)
:vector(v)
{
  index = ind;
}

template<typename T>
inline TDynamicVectorIterator<T>::TDynamicVectorIterator(TDynamicVectorIterator<T>& iv)
:vector(iv.vector), index(iv.index)
{
}

template<typename T>
inline bool TDynamicVectorIterator<T>::operator==(const TDynamicVectorIterator<T>& v)
{
  return (&vector == &(v.vector) && index == v.index);
}

template<typename T>
inline bool TDynamicVectorIterator<T>::operator!=(const TDynamicVectorIterator<T>& v)
{
  return !(this->operator==(v));
}

template<typename T>
inline TDynamicVectorIterator<T> TDynamicVectorIterator<T>::operator++()
{
  TDynamicVectorIterator<T> res(*this);
  this->index++;
  res.index++;
  if (res.index > res.vector.size())
    res.index = res.vector.size();
  return res;
}

template<typename T>
inline TDynamicVectorIterator<T> TDynamicVectorIterator<T>::operator--()
{
  TDynamicVectorIterator<T> res(*this);
  res.index--;
  if (res.index < 0)
    res.index = 0;
  return res;
}

template<typename T>
inline T& TDynamicVectorIterator<T>::operator*()
{
  return vector.at(index);
}
```

8.2 Реализация класса квадратных матриц

```
#pragma once
#include "Vector.h"

template <class T>
class TSquareMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TSquareMatrix(size_t size1);
  TSquareMatrix(size_t size,const T* arr);
  TSquareMatrix(const TSquareMatrix<T>& mat) noexcept;
  TSquareMatrix(TSquareMatrix<T>&& mat);

  bool operator == (const TSquareMatrix<T>& mat) const noexcept;
  bool operator != (const TSquareMatrix<T>& mat) const noexcept;

  TSquareMatrix<T> operator + (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator - (const TSquareMatrix<T>& mat);
  TSquareMatrix<T> operator * (const TSquareMatrix<T>& mat);

  TSquareMatrix<T> operator = (TSquareMatrix<T>&& mat);
  TSquareMatrix<T> operator = (const TSquareMatrix<T>& mat);
};

template<class T>
inline TSquareMatrix<T>::TSquareMatrix(size_t size, const T* arr) :
TSquareMatrix<T>(size)
{
  if (size == 0)
    throw out_of_range("size must be greater than zero");
  for (size_t i = 0; i < sz*sz; i++)
    pMem[i / sz][i % sz] = arr[i];
}

template<class T>
inline TSquareMatrix<T>::TSquareMatrix(size_t size1): TDynamicVector<TDynamicVector<T>>
(size1)
{
  sz = size1;
  for (size_t i = 0; i < sz; i++)
    pMem[i] = TDynamicVector<T>(sz);
}

template<class T>
inline TSquareMatrix<T>::TSquareMatrix(const TSquareMatrix<T>& mat) noexcept :
TDynamicVector<TDynamicVector<T>> (mat)
{
  if (mat.pMem == nullptr)
  {
    pMem = nullptr;
    sz = 0;
  }
  else
  {
    sz = mat.sz;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < sz; i++)
      pMem[i] = mat.pMem[i];
  }
}
```

```
template<class T>
inline TSquareMatrix<T>::TSquareMatrix(TSquareMatrix<T>&& mat)
{
  pMem = mat.pMem;
  sz = mat.sz;
  mat.pMem = nullptr;
  mat.sz = 0;
}

template<class T>
inline bool TSquareMatrix<T>::operator==(const TSquareMatrix<T>& mat) const noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator ==(mat);
}

template<class T>
inline bool TSquareMatrix<T>::operator!=(const TSquareMatrix<T>& mat) const noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator !=(mat);
}

template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator+(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] + mat.pMem[i];
  return res;
}

template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator-(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] - mat.pMem[i];
  return res;
}

template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator*(const TSquareMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TSquareMatrix<T> res(sz);
  for (size_t row = 0; row < sz; row++)
    for (size_t col = 0; col < sz; col++)
    {
      T sum = 0;
      for (size_t k = 0; k < sz; k++)
        sum += pMem[row][k] * mat.pMem[k][col];
      res.pMem[row][col] = sum;
    }
  return res;
}

template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator=(TSquareMatrix<T>&& mat)
{
  if (this != &mat)
```

```
  {
    if (pMem != nullptr)
      delete[] pMem;

    pMem = mat.pMem;
    sz = mat.sz;
    mat.pMem = nullptr;
    mat.sz = 0;
  }
  else
    throw "copy itself";
  return *this;
}

template<class T>
inline TSquareMatrix<T> TSquareMatrix<T>::operator=(const TSquareMatrix<T>& mat)
{
  if (this != &mat)
  {
    sz = mat.sz;
    if (pMem != nullptr)
      delete[] pMem;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < mat.sz; i++)
      pMem[i] = mat.pMem[i];
  }
  else
    throw "copy itself";
  return *this;
}

template<class T>
ostream& operator <<(ostream& ostr, TSquareMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j < p.size(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }

  return ostr;
}

template<class T>
istream& operator >>(istream& istr, TSquareMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TSquareMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;
}
```

8.3 Реализация класса треугольных матриц

```
#pragma once
#include "Vector.h"

template <class T>
class TTrangleMatrix : public TDynamicVector<TDynamicVector<T>>
{
  using::TDynamicVector<TDynamicVector<T>>::pMem;
  using::TDynamicVector<TDynamicVector<T>>::sz;
public:
  TTrangleMatrix(size_t size1);
  TTrangleMatrix(size_t size, const T* arr);
  TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept;
  TTrangleMatrix(TTrangleMatrix<T>&& mat);

  bool operator == (const TTrangleMatrix<T>& mat) const noexcept;
  bool operator != (const TTrangleMatrix<T>& mat) const noexcept;

  TTrangleMatrix<T> operator + (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator - (const TTrangleMatrix<T>& mat);
  TTrangleMatrix<T> operator * (const TTrangleMatrix<T>& mat);

  TTrangleMatrix<T> operator = (TTrangleMatrix<T>&& mat);
  TTrangleMatrix<T> operator = (const TTrangleMatrix<T>& mat);
};

template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(size_t size1) :
TDynamicVector<TDynamicVector<T>>(size1)
{
  sz = size1;
  for (size_t i = 0; i < sz; i++)
    pMem[i] = TDynamicVector<T>(i+1);
}

template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(size_t size, const T* arr) :
TTrangleMatrix<T>(size)
{
  if (size == 0)
    throw out_of_range("size must be greater than zero");
  size_t iter = 0;
  for (size_t i = 0; i < sz; i++)
    for (int j = 0; j <= i; j++)
    {
      pMem[i][j] = arr[iter];
      iter++;
    }
}

template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(const TTrangleMatrix<T>& mat) noexcept
{
  if (mat.pMem == nullptr)
  {
    pMem = nullptr;
    sz = 0;
  }
  else
  {
    sz = mat.sz;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < sz; i++)
      pMem[i] = mat.pMem[i];
```

```
  }
}

template<class T>
inline TTrangleMatrix<T>::TTrangleMatrix(TTrangleMatrix<T>&& mat)
{
  pMem = mat.pMem;
  sz = mat.sz;
  mat.pMem = nullptr;
  mat.sz = 0;
}

template<class T>
inline bool TTrangleMatrix<T>::operator==(const TTrangleMatrix<T>& mat) const noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator ==(mat);
}

template<class T>
inline bool TTrangleMatrix<T>::operator!=(const TTrangleMatrix<T>& mat) const noexcept
{
  return TDynamicVector<TDynamicVector<T>>::operator !=(mat);
}

template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator+(const TTrangleMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TTrangleMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] + mat.pMem[i];
  return res;
}

template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator-(const TTrangleMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TTrangleMatrix<T> res(sz);
  for (size_t i = 0; i < sz; i++)
    res.pMem[i] = pMem[i] - mat.pMem[i];
  return res;
}

template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator*(const TTrangleMatrix<T>& mat)
{
  if (sz != mat.sz)
    throw "different sizes";
  TTrangleMatrix<T> res(sz);
  for (size_t row = 0; row < sz; row++)
    for (size_t col = 0; col <= row; col++)
    {
      T sum = 0;
      for (size_t k = col; k <= row; k++)
        sum += pMem[row][k] * mat.pMem[k][col];
      res.pMem[row][col] = sum;
    }
  return res;
}

template<class T>
```

```
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator=(TTrangleMatrix<T>&& mat)
{
  if (this != &mat)
  {
    if (pMem != nullptr)
      delete[] pMem;

    pMem = mat.pMem;
    sz = mat.sz;
    mat.pMem = nullptr;
    mat.sz = 0;
  }
  else
    throw "copy itself";
  return *this;
}

template<class T>
inline TTrangleMatrix<T> TTrangleMatrix<T>::operator=(const TTrangleMatrix<T>& mat)
{
  if (this != &mat)
  {
    sz = mat.sz;
    if (pMem != nullptr)
      delete[] pMem;
    pMem = new TDynamicVector<T>[sz];
    for (size_t i = 0; i < mat.sz; i++)
      pMem[i] = mat.pMem[i];
  }
  else
    throw "copy itself";
  return *this;
}

template<class T>
ostream& operator <<(ostream& ostr, TTrangleMatrix<T>& p)
{
  for (size_t i = 0; i < p.size(); i++)
  {
    auto v = p[i];
    for (size_t j = 0; j <= i; j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr << endl;
  }
  return ostr;
}

template<class T>
istream& operator >>(istream& istr, TTrangleMatrix<T>& p)
{
  size_t size;
  istr >> size;
  TTrangleMatrix<T> istrmatrix(size);
  for (size_t i = 0; i < size; i++)
    for (size_t j = 0; j < size; j++)
      istr >> istrmatrix[i][j];
  p = istrmatrix;
  return istr;
}
```