МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Национальный исследовательский**

**Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

Отчёт по лабораторной работе

Выполнил:

студент группы 3821Б1ПМ3

Мартынов А.Ю.

Проверил:

Заведующий лабораторией суперкомпьютерных технологий и высокопроизводительных вычислений

Лебедев И.Г.

Нижний Новгород

2022 г.

**Оглавление**

[3. Руководство программиста 14](#_Toc122378565)

[3.1 Описание структур данных 14](#_Toc122378566)

[3.1.1 Классы 14](#_Toc122378567)

[3.1.2 Библиотеки 19](#_Toc122378568)

[3.1.3 Типы данных 20](#_Toc122378569)

[3.2 Описание алгоритмов 21](#_Toc122378570)

[4. Эксперименты 24](#_Toc122378571)

[5. Заключение 27](#_Toc122378572)

[6. Литература 28](#_Toc122378573)

[7. Приложения 29](#_Toc122378574)

[7.1 Реализация заголовочного файла «tmatrix.h» 29](#_Toc122378575)

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.

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 С++ 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. На данном простом примере необходимо будет разобраться с принципом его работы и использованием при обходе структур данных. Для проверки используются Google-tests. Они позволяют убедиться, что написанные классы функционируют правильно.

В данной лабораторной работе для изучения особенностей работы с классами, была поставлена задача: на языке «С++», 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.

* 1. **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. Классы матриц должны быть наследниками класса вектора и по сути являться векторами векторов.

1. **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. (См. Рис. 1)

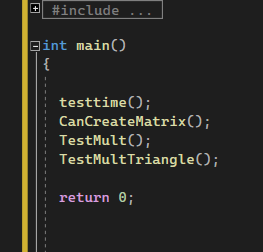


Рисунок 1. Список тестовых функций.

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)

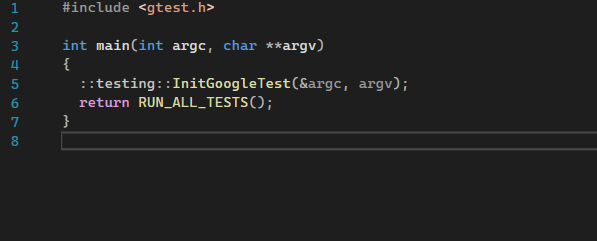


Рисунок 2. Основной файл с тестами.

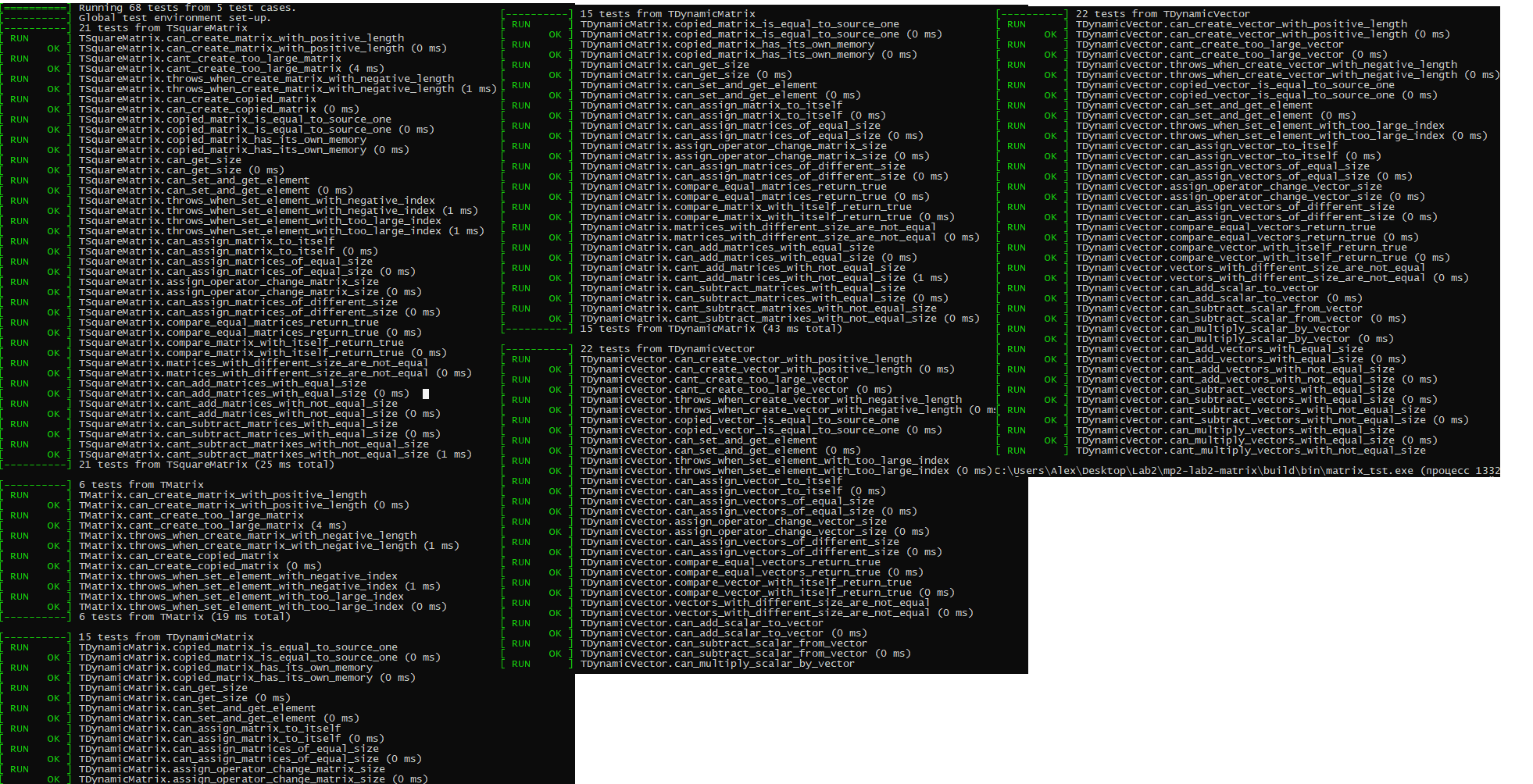


Рисунок 3. Вид консоли при выполнении Гугл-тестов

Так как исходные файлы, содержащие описания классов вынесены в статистическую библиотеку, тонаписанный 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 данный файл 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.

* 1. **Класс векторов**

Чтобы подключить библиотеку с вектором, необходимо в начале программы написать «#include "TMatrix.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 | 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 | Возвращаемое значение |
| 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& operator=(const TDynamicVector& v) | Перегрузка оператора присваивания, которая переопределяет вектор и присваивает ему нужное значение | Ссылка на вектор |
| TDynamicVector& operator=(TDynamicVector&& v) noexcept | Перегрузка оператора присваивания с перемещением, которая переопределяет вектор путем переопределения указателя на область памяти, а перемещаемые данные удаляются | Ссылка на вектор |
| TDynamicVector operator+(const TDynamicVector& v) | Перегрузка оператора «+», при которой происходит покоординатное сложение векторов (если длины векторов разные то программа выдаст исключение) | Вектор |
| TDynamicVector operator-(const TDynamicVector& v) | Перегрузка оператора «-», при которой происходит покоординатное вычитание векторов (если длины векторов разные то программа выдаст исключение) | Вектор |
| T operator\*(const TDynamicVector& v) noexcept(noexcept(T())) | Перегрузка оператора «\*», которая отвечает за вычисление скалярного произведения двух векторов (если длины векторов разные то программа выдаст исключение) | Т |
| TDynamicVector operator+(T val) | Перегрузка оператора «+», при которой на вход подается шаблонный элемент и он прибавляется к каждой координате вектора | Вектор |
| TDynamicVector operator-(T val) | Перегрузка оператора «-», при которой на вход подается шаблонный элемент и он вычитается из каждой координаты 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 ostream |
| friend ostream& operator>>(ostream& ostr, const TDynamicVector& v) | Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер вектора, после чего ввести каждую координату вектора | Ссылку на istream |

* 1. **Класс квадратных матриц**

Чтобы подключить библиотеку с матрицами, необходимо в начале программы написать «#include "TMatrix.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 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:

|  |  |  |
| --- | --- | --- |
| Методы | Описание | Возвращаемое значение |
| bool operator==(const TSquareMatrix & mt) 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 |
| TDynamicMatrix operator=(const TDynamicMatrix<T>& mt) | 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 operator+ (const TSquareMatrix& mt); | Перегрузка оператора «+», при которой выполняется поэлементное сложение | Матрица |
| TSquareMatrix operator+ (const TSquareMatrix& mt); | Перегрузка оператора «-», при которой выполняется поэлементное вычитание | Матрица |
| TSquareMatrix<T> operator\*(const TSquareMatrix<T>& mt) | 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 |
| 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 |
| friend ostream& operator<<(ostream& out, const TSquareMatrix& mt) | Перегрузка оператора потокового вывода, красиво выводит матрицу на консоль | Ссылка на объект ostream |
| friend istream& operator>>(istream& in, TSquareMatrix& mt) | 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 3

* 1. **Класс треугольных матриц**

|  |  |
| --- | --- |
| TDynamicMatrix(size\_t size1)  Пример:  TDynamicMatrix<int> M(5) | Конструктор, принимающий на вход количество строк и столбцов в матрице, а также заполняющий её нулями. |
| TDynamicMatrix(const TDynamicVector<TDynamicVector<T> >& mt) :TDynamicVector<TDynamicVector<T>> (mt) {} | Конструктор копирования, который на вход принимает другую “матрицу” – вектор векторов и копирует данные в новую матрицу. |
| TDynamicMatrix( TDynamicMatrix<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. |

Чтобы подключить библиотеку с нижне-треугольными матрицами, необходимо в начале программы написать «#include "TMatrix.h"», после чего пользоваться инструментарием данного класса. Создать матрицу можно тремя способами, тип данных, который будет использоваться матрицей, указывается в треугольных скобках

#### Таблица 4.

#### 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 TDynamicMatrix <T>& mt) 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 TDynamicMatrix <T>& mt) 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 |
| TDynamicMatrix <T> operator + (const TDynamicMatrix <T>& mt) | Перегрузка оператора «+», при которой выполняется поэлементное сложение | Матрица |
| TDynamicMatrix <T> operator - (const TDynamicMatrix <T>& mt) | Перегрузка оператора «-», при которой выполняется поэлементное вычитание | Матрица |
| TDynamicMatrix <T> operator \* (const TDynamicMatrix <T>& mt) | 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 |
| TDynamicMatrix <T> operator = (TDynamicMatrix <T>&& mt) | Перегрузка оператора присваивания с перемещением, которая переопределяет матрицу | Матрица |

|  |  |  |
| --- | --- | --- |
|  | и присваивает ей нужное значение, удаляя данные из места, откуда производилось копирование |  |
| TDynamicMatrix <T> operator = (const TDynamicMatrix <T>& mat) | Перегрузка оператора присваивания, которая переопределяет матрицу и присваивает ей нужное значение | Матрица |
| friend ostream& operator<<(ostream& out, const TDynamicMatrix& mt) | Перегрузка оператора потокового вывода, красиво выводит матрицу на консоль | Ссылка на объект ostream |
| friend ostream& operator>>(ostream& out, const TDynamicMatrix& mt) | 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 5.

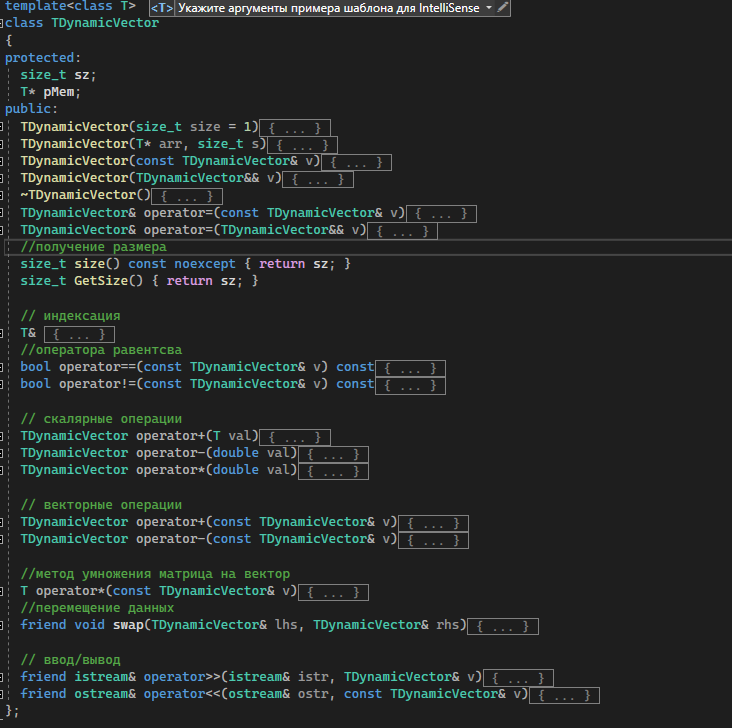
# Руководство программиста

## Описание структур данных

### Классы

**Класс векторов:**

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



#### Рисунок 4. Класс векторов. Поля класса:

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

#### «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций

#### «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.

#### «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.

#### «~TDynamicVector», деструктор, очищающий выделенную для массива память в конце работы программы.

#### «size()», метод, возвращающий размер вектора.

#### Перегрузка оператора «[]» реализована с проверкой, в случае если мы введём неподходящий индекс, то это вызовёт исключение

#### Реализованы операторы для арифметический операций, а именно (\*,

#### +, -, =, ==, !=), принимающие на вход константные ссылки на вектора либо же шаблонный элемент.

#### Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.

#### 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)

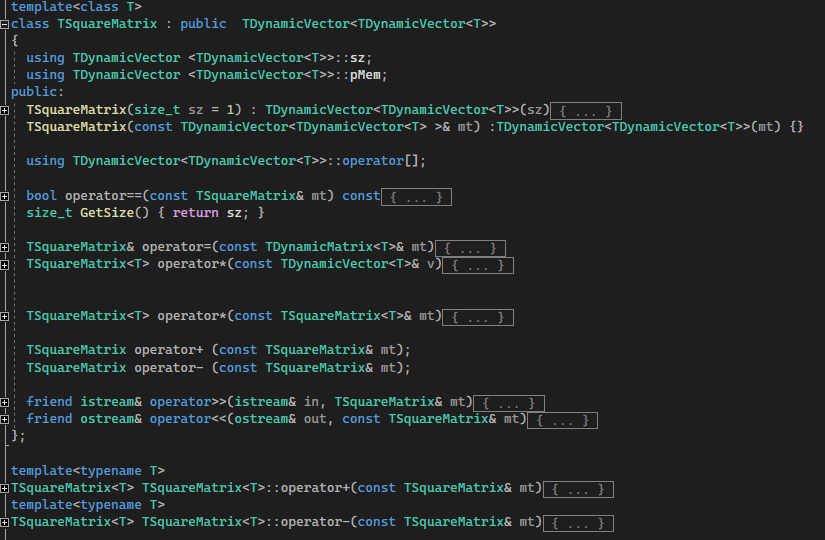
#### 

#### Рисунок 5. Реализация перегрузок потокового ввода и вывода векторов.

**Класс квадратных матриц:**

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

#### (См. Рис.6)



#### Рисунок 6. Класс квадратных матриц. Поля класса:

#### У класса нет как таковых полей, вместо этого используются поля класса-предка, ведь матрица в данном случае - вектор, состоящий из векторов. Данное не очень рекомендуется для реализации, тк учитывая ISA CPU, то данные в DataPat в кэше будут храниться линейно, а тк у нас данные будут лежать не в линейном порядке, то тогда процессор не сможет их павильно подгрузить, из-за чего будут просадки по скорости работы проги. Также, если учитывать, что рассматриваются квадратные матрицы, то можно не вводить вторую переменную для размера.

#### «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций

#### «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.

#### «TSquareMatrix», это 4 конструктора, один совмещённый, по умолчанию и инициализатор, принимающий на вход размер квадратной матрицы, два, это конструкторы копирования и перемещения, первый копирует данные, а другой передаёт память новому объекту, а старую уничтожает

#### Также дальше в описании класса идут перегрузки операций, а именно (\*,

#### +, -, =, ==, !=), принимающие на вход константные ссылки на вектора.

#### Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.

#### 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)

#### Деструктор отдельно не реализуется в матрицах, так как он неявно наследуется от класса вектора.

#### 

#### Рисунок 7. Реализация перегруженных оператор ввода/вывода

**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)

#### 

#### 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.

#### «pMem», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций

#### «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.

#### «TDynamicMatrix», 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

#### Также дальше в описании класса идут перегрузки операций, а именно (\*,

#### +, -, =, ==, !=), принимающие на вход константные ссылки на вектора.

#### Оператор присваивания с перемещением работает примерно также, как и стандартный присваивания, но очищает исходный объект.

#### 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)

#### Деструктор отдельно не реализуется в матрицах, так как он неявно наследуется от класса вектора.

#### 

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

### Библиотеки

#### В данной программе используется несколько библиотек:

#### «iostream», основная библиотека, включающая в себя элементы работы с потоковым вводом и выводом, а также содержащая базовые элементы.

#### «fstream», расширенная библиотека для работы с потоковыми данными, позволяющая создать переменную потокового типа, в которую записывать данные, чтобы проводить тесты.

#### «time.h» и «stdlib.h» используются для подсчета времени работы алгоритма для эксперимента.

#### «../gtest/gtest.h», библиотека с гугл-тестами.

### 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:

#### «TDynamicVector<T>», образующая объект, вектор шаблонного типа «Т».

#### «TSquareMatrix<Т>», образующая объект, квадратную матрицу шаблонного типа «Т».

#### «TTrangleMatrix<Т>», образующая объект, треугольную матрицу шаблонного типа «Т».

#### «TVectorIterator<T>», образующая объект, вектор-итератор шаблонного типа

#### «Т».

#### «int», создающий переменную целочисленного типа, которая может в дальнейшем понадобиться в расчётах.

#### «bool», логический тип данных, имеющий только 2 значения, истина или ложь.

#### «stringstream», универсальный потоковый тип данных, который можно использовать и для ввода, и для вывода.

#### «double», используемый для создания вещественного типа данных.

#### «Т» 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.

#### «ostream», тип данных, для работы с потоковым выводом.

#### «istream», тип данных для работы с потоковым вводом

#### «size\_t», переменная, используемая для объявления неотрицательных переменных.

#### Некоторые переменные могут быть заранее объявлены как «const», для того чтобы они были неизменными.

#### Также используются другие стандартные для языка «С++» типы данных, которые не нуждаются в отдельном описании.

## 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. Умножение 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\*nсложность выполнения оценивается теоретически как 𝑂(𝑛3).

𝑂(𝑛 )

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

#### 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.

#### 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 1

#### 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++)

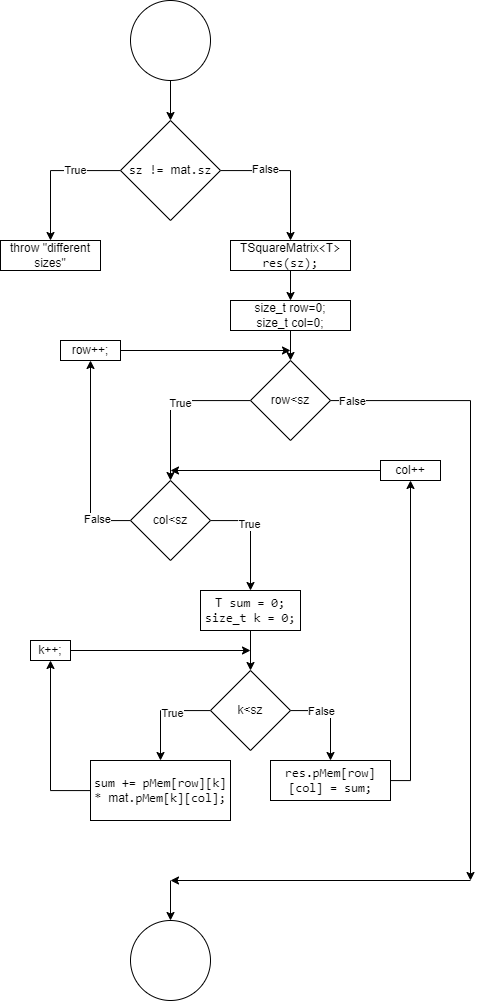
sum += pMem[row][k] \* mat.pMem[k][col]; res.pMem[row][col] = sum;

}

return res;

#### Фрагмент кода 2.

#### В этом и состоит самый интересный и трудоемкий алгоритм, представленный в данной лабораторной работе (См. Рис.10).



#### Рисунок 10. Блок-схема алгоритма умножения матриц.

# 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 3.

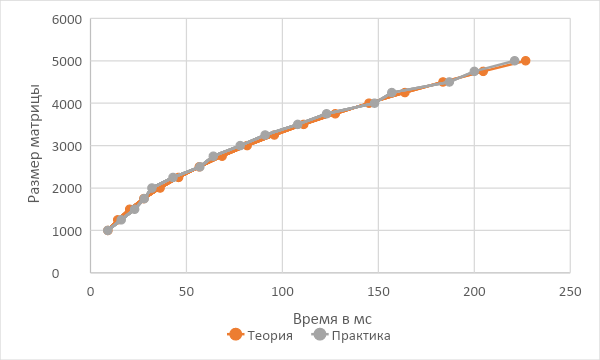
#### Сначала запустим этот тест размера 1000 и посмотрим, сколько же времени затратит данный алгоритм. (См. Рис.12)



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

#### Сложение матриц оценивается как 𝑂(𝑛2), так же, как и выделение памяти для создания временной матрицы. Теоретически время будет равно 𝑇(𝑛)≈𝑎𝑛2 . Подставив

#### наши данные, получаем, что 𝑎≈9, 07 • 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.

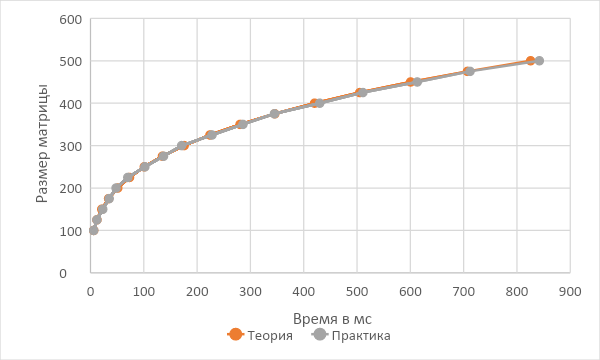
#### Получив практические данные по времени для каждой размерности матриц, мы получили график и теперь можем сравнить. Можно видеть, что практические данные не сильно отличаются от теоретически предполагаемых, есть конечно несколько отклонений, но в целом все данные сходятся.

#### 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 n2 элементов, требующее 𝑏𝑛2 времени. Коэффициенты 𝑎 и 𝑏 несложно найти из имеющихся данных: a = 6,75\*10-6 b = -7,24 \* 10-5 получаем, что кривая времени от размера задаётся через an3+bn2

𝑛

−6 −5

𝑂(𝑛 )



#### График 2.

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

# Заключение

#### В результате проведенной лабораторной работы, на языке программирования

#### «С++» 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.

# Литература

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

#### Бьерн Страуструп. Язык программирования С++ - Режим доступа: <http://8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf>

# Приложения

# Реализация заголовочного файла «tmatrix.h»

#ifndef \_\_TDynamicMatrix\_H\_\_

#define \_\_TDynamicMatrix\_H\_\_

#include <iostream>

#include<fstream>

using namespace std;

const int MAX\_VECTOR\_SIZE = 100000000;

const int MAX\_MATRIX\_SIZE = 10000;

// Динамический вектор -

// шаблонный вектор на динамической памяти

template<class T>

class TDynamicVector

{

protected:

size\_t sz;

T\* pMem;

public:

TDynamicVector(size\_t size = 1) : sz(size)

{

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

throw out\_of\_range("Vector size should be greater than zero");

pMem = new T[sz]();// {}; // У типа T д.б. констуктор по умолчанию

}

TDynamicVector(T\* arr, size\_t s) : sz(s)

{

assert(arr != nullptr && "TDynamicVector ctor requires non-nullptr arg");

pMem = new T[sz];

std::copy(arr, arr + sz, pMem);

}

TDynamicVector(const TDynamicVector& v)

{

if (v.sz <= 0) throw "size of v <= 0";

sz = v.sz;

pMem = new T[sz];

for (size\_t i = 0; i < sz; i++)

pMem[i] = v.pMem[i];

}

TDynamicVector(TDynamicVector&& v) noexcept

{

sz = v.sz;

pMem = v.pMem;

v.sz = 0;

v.pMem = nullptr;

}

~TDynamicVector()

{

if (pMem != nullptr)

delete[] pMem;

pMem = nullptr;

}

TDynamicVector& operator=(const TDynamicVector& v)

{

if (this != &v)

{

if (pMem != nullptr)

{

delete[] pMem;

pMem = nullptr;

sz = 0;

}

if (v.pMem != nullptr && v.sz > 0)

{

sz = v.sz;

pMem = new T[sz];

for (size\_t i = 0; i < sz; i++)

pMem[i] = v.pMem[i];

}

else

throw "vector v = nullptr or size = 0";

}

return \*this;

}

TDynamicVector& operator=(TDynamicVector&& v) noexcept

{

if (this != &v)

{

if (pMem != nullptr)

{

delete[] pMem;

pMem = nullptr;

sz = 0;

}

sz = v.sz;

pMem = v.pMem;

v.sz = 0;

v.pMem = nullptr;

}

return \*this;

}

//получение размера

size\_t size() const noexcept { return sz; }

size\_t GetSize() { return sz; }

// индексация

T& operator[](size\_t ind)

{

if (pMem == nullptr) throw "pMem = nullptr";

if (ind < 0 || ind >= sz) throw "the index less or greater than size of vector ";

else

return pMem[ind];

}

//оператора равентсва

bool operator==(const TDynamicVector& v) const noexcept

{

if (this == &v)

return true;

if (sz != v.sz)

return false;

for (size\_t i = 0; i < sz; i++) {

if (pMem[i] != v.pMem[i])

return false;

}

return true;

}

bool operator!=(const TDynamicVector& v) const noexcept

{

return !(\*this == v);

}

// скалярные операции

TDynamicVector operator+(T val)

{

TDynamicVector<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = pMem[i] + val;

return result;

}

TDynamicVector operator-(double val)

{

TDynamicVector<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = pMem[i] - val;

return result;

}

TDynamicVector operator\*(double val)

{

TDynamicVector<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = pMem[i] \* val;

return result;

}

// векторные операции

TDynamicVector operator+(const TDynamicVector& v)

{

if (sz != v.sz) throw "sizes not equals";

TDynamicVector<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = pMem[i] + v.pMem[i];

return result;

}

TDynamicVector operator-(const TDynamicVector& v)

{

if (sz != v.sz) throw "sizes not equals";

TDynamicVector<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = pMem[i] - v.pMem[i];

return result;

}

//метод умножения матрица на вектор

T operator\*(const TDynamicVector& v) noexcept(noexcept(T()))

{

if (sz != v.sz) throw "Different size of TVector's";

T res = 0;

if (&v != nullptr)

{

for (size\_t i = 0; i < sz; i++)

{

res = pMem[i] \* v.pMem[i];

}

}

return res;

}

//перемещение данных

friend void swap(TDynamicVector& lhs, TDynamicVector& rhs) noexcept

{

std::swap(lhs.sz, rhs.sz);

std::swap(lhs.pMem, rhs.pMem);

}

// ввод/вывод

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;

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Динамическая матрица -

// шаблонная матрица на динамической памяти

template<class T>

class TDynamicMatrix : public TDynamicVector<TDynamicVector<T>>

{

using TDynamicVector<TDynamicVector<T>>::pMem;

using TDynamicVector<TDynamicVector<T>>::sz;

public:

TDynamicMatrix(size\_t s = 1) : TDynamicVector<TDynamicVector<T>>(s)

{

if (sz > MAX\_MATRIX\_SIZE) throw "GG";

for (size\_t i = 0; i < sz; i++)

pMem[i] = TDynamicVector<T>(sz);

}

TDynamicMatrix(const TDynamicVector<TDynamicVector<T> >& mt) :TDynamicVector<TDynamicVector<T>> (mt) {}

TDynamicMatrix(TTrangleMatrix<T>&& mat)

using TDynamicVector<TDynamicVector<T>>::operator[];

// сравнение

bool operator==(const TDynamicMatrix& mt) const noexcept

{

return TDynamicVector<TDynamicVector<T>>:: operator==(mt);

}

const

size\_t GetSize() { return sz; }

TDynamicMatrix operator=(const TDynamicMatrix<T>& mt)

{

if (this == &mt)

return \*this;

TDynamicVector<T>\* tmp = new TDynamicVector<T>[mt.sz];

delete[] this->pMem;

this->pMem = tmp;

this->sz = mt.sz;

for (int i = 0; i < this->sz; i++) {

this->pMem[i] = mt.pMem[i];

}

return \*this;

}

// матрично-векторные операции

TDynamicMatrix<T> operator\*(const TDynamicVector<T>& v)

{

TDynamicMatrix<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = 0;

for (size\_t i = 0; i < sz; i++)

for (size\_t j = 0; j <= i; j++)

{

result.pMem[i] += pMem[i][j] \* m.pMem[i][j];

}

return result;

}

TDynamicMatrix operator+ (const TDynamicMatrix& mt);

TDynamicMatrix operator- (const TDynamicMatrix& mt);

// ввод/вывод

friend istream& operator>>(istream& in, TDynamicMatrix& mt)

{

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

in >> mt.pMem[i];

return in;

}

friend ostream& operator<<(ostream& out, const TDynamicMatrix& mt)

{

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

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

return out;

}

};

template<typename T>// сравнение

TDynamicMatrix<T> TDynamicMatrix<T>::operator+(const TDynamicMatrix& mt)

{

return TDynamicVector<TDynamicVector<T>>:: operator+(mt);

}

template<typename T>// сравнение

TDynamicMatrix<T> TDynamicMatrix<T>::operator-(const TDynamicMatrix& mt)

{

return TDynamicVector<TDynamicVector<T>>:: operator-(mt);

}

template<class T>

class TSquareMatrix : public TDynamicVector<TDynamicVector<T>>

{

using TDynamicVector <TDynamicVector<T>>::sz;

using TDynamicVector <TDynamicVector<T>>::pMem;

public:

TSquareMatrix(size\_t sz = 1) : TDynamicVector<TDynamicVector<T>>(sz)

{

if (sz > MAX\_MATRIX\_SIZE) throw "GG";

this->sz = sz;

for (size\_t i = 0; i < this->sz; i++)

pMem[i] = TDynamicVector<T>(sz);

}

TSquareMatrix(const TDynamicVector<TDynamicVector<T> >& mt) :TDynamicVector<TDynamicVector<T>>(mt) {}

using TDynamicVector<TDynamicVector<T>>::operator[];

bool operator==(const TSquareMatrix& mt) const noexcept

{

return TDynamicVector<TDynamicVector<T>>:: operator==(mt);

}

size\_t GetSize() { return sz; }

TSquareMatrix& operator=(const TDynamicMatrix<T>& mt)

{

if (this == &mt) return this;

if (sz != mt.sz) throw"GG";

TDynamicVector<T>\* tmp = new TDynamicVector<T>[mt.sz];

delete[] this->pMem;

this->pMem = tmp;

this->sz = mt.sz;

for (int i = 0; i < this->sz; i++) {

this->pMem[i] = mt.pMem[i];

}

return this;

}

TSquareMatrix<T> operator\*(const TDynamicVector<T>& v)

{

TSquareMatrix<T> result(\*this);

for (size\_t i = 0; i < sz; i++)

result.pMem[i] = 0;

for (size\_t i = 0; i < sz; i++)

for (size\_t j = 0; j <= i; j++)

{

result.pMem[i] += pMem[i][j] \* m.pMem[i][j];

}

return result;

}

TSquareMatrix<T> operator\*(const TSquareMatrix<T>& mt)

{

T sum = 0;

if (sz != mt.sz)

throw "different sizes";

TSquareMatrix<T> res(sz);

for (size\_t row = 0; row < sz; row++)

for (size\_t col = 0; col < sz; col++)

{

sum = 0;

for (size\_t k = 0; k < sz; k++)

sum += pMem[row][k] \* mt.pMem[k][col];

res.pMem[row][col] = sum;

}

return res;

}

TSquareMatrix operator+ (const TSquareMatrix& mt);

TSquareMatrix operator- (const TSquareMatrix& mt);

friend istream& operator>>(istream& in, TSquareMatrix& mt)

{

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

in >> mt.pMem[i];

return in;

}

friend ostream& operator<<(ostream& out, const TSquareMatrix& mt)

{

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

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

return out;

}

};

template<typename T>

TSquareMatrix<T> TSquareMatrix<T>::operator+(const TSquareMatrix& mt)

{

return TDynamicVector<TDynamicVector<T>>:: operator+(mt);

}

template<typename T>

TSquareMatrix<T> TSquareMatrix<T>::operator-(const TSquareMatrix& mt)

{

return TDynamicVector<TDynamicVector<T>>:: operator-(mt);

}

#endif