Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

Высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики механики

Матрицы

Отчет по лабораторной работе

Выполнил:

студент ИИТММ гр. 381706-2

Антипин А.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Нижний Новгород

2018 г.

**Содержание**

[Введение 3](#_Toc532933808)

[Постановка целей и задач 4](#_Toc532933809)

[Руководство пользователя 5](#_Toc532933810)

[Руководство программиста 6](#_Toc532933811)

[Описание структуры программы 6](#_Toc532933812)

[Описание структур данных 7](#_Toc532933813)

[Описание алгоритмов 7](#_Toc532933814)

[Эксперименты 10](#_Toc532933815)

[Результаты работы программы 10](#_Toc532933816)

[Время выполнения 11](#_Toc532933817)

[Заключение 13](#_Toc532933818)

[Литература 14](#_Toc532933819)

[Приложение 15](#_Toc532933820)

# Введение

Матрицы являются одним из самых удобных способов представления информации. Они применяются в математике для компактной записи систем линейных алгебраических или дифференциальных уравнений. Также они нашли широкое применение в программировании, так как матричный способ представления графической информации является наиболее простым и удобным, если не единственным наиболее правильным.

Операции, определенные над матрицами, также обеспечивают хорошее быстродействие программы и достаточно легко реализуются в программировании. Этими операциями являются: сложение и вычитание матриц одинакового размера, умножение матриц друг на друга и на скаляр.

Помимо матриц общего вида, для которых наиболее естественной и наиболее часто используемой представляется программная реализация в виде двумерного массива, в математических приложениях выделяются различные матрицы специальных видов (треугольные, диагональные и т.д.). Для таких матриц предпочтительно создание собственных способов хранения и обработки, учитывающих специфику их структуры, и потому более эффективных.

В данной работе будут рассмотрены верхнетреугольные матрицы, которые представляют собой квадратные матрицы, элементы которой ниже главной диагонали равны нулю.

# Постановка целей и задач

Основной целью лабораторной работы является создание программных средств хранения и обработки верхнетреугольных матриц. Средства обработки включают в себя выполнение матричных операций, таких как:

* сложение и вычитание двух матриц
* умножение двух матриц
* умножение матрицы на скаляр

Для реализации алгоритмов будет использоваться 2 шаблонных класса:

* Vector
* Matrix

Для проверки правильности работы этих классов будут написаны тесты с использованием фреймворка Google Test, а также тестовый образец программы, которая использует данный класс матриц.

# Руководство пользователя

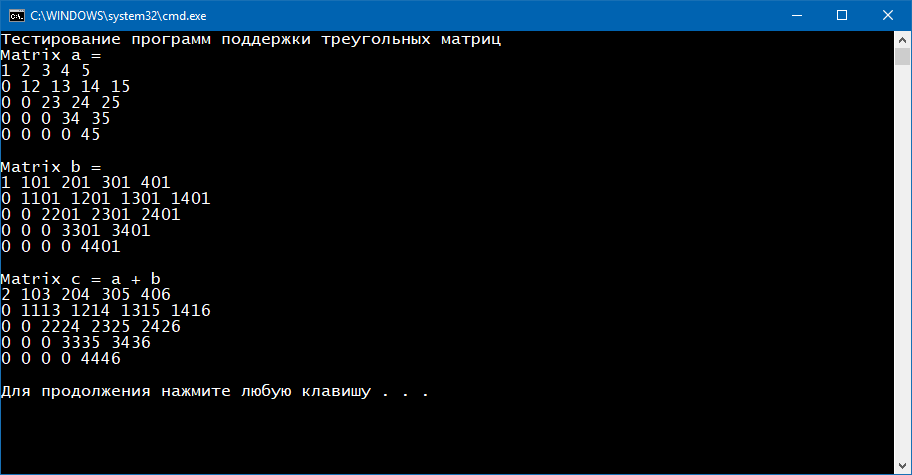
После запуска программы пользователя встречает консольное окно (рис. 1):

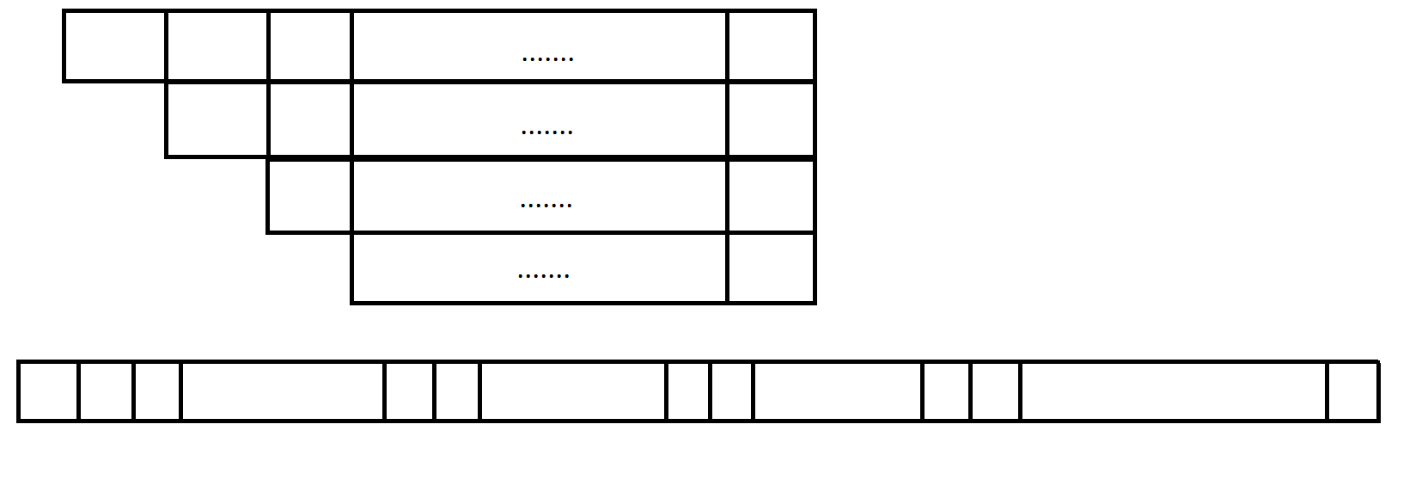
рис. 1 (вывод сложения 2 матриц для пользователя)

в котором выведены две матрицы: a и b, а также матрица c, являющаяся результатом сложения этих двух матриц.

# Руководство программиста

## Описание структуры программы

Матрица будем реализовывать с помощью набора векторов:



То есть для реализации алгоритмов будет использовано 2 класса:

* Класс «Вектор» (Vector).
* Класс «Матрицы» (Matrix), который будет использовать класс векторов.

А также проект использующий фреймворк Google Test, для проверки правильности работы этих классов и тесовый проект, который будет показываться пользователю.

Древо классов

**Класс gtest**

**test\_main.cpp**

**vector\_test.cpp**

**matrix\_test.cpp**

**Проект для пользователя**

**main.cpp**

**matrix.h**

**Класс matrix**

**vector.h**

**Класс vector**

**Класс Vector:**

Класс вектор задает работу с классом matrix посредством возможности создания строк этой матрицы и работы с отдельными элементами этого массива. Также можно выполнять операции, связанные с векторами. Такие как сложение, вычитание векторов, скалярное произведение и др.

**Класс Matrix:**

Класс matrix содержит реализацию работы с верхнетреугольными матрицами. В нем реализованы такие методы, как сложить, вычесть, перемножить 2 матрицы.

**Класс gtest:**

Класс gtest реализует тестирование классов vector и matrix, по средствам фреймворка Google Test. Тесты пишутся для каждого метода классов, каждого ветвления этих методов и для всех возможных исключений этих методов. В файле «vector\_test.cpp» реализованы тесты для класса vector, а в файле «matrix\_testcpp» соответственно для класса matrix.

**Проект Vector:**

В данном проекте реализован примет использования верхнетреугольных матриц, который в конечном итоге и будет доступен пользователю.

## Описание структур данных

**Класс Vector:**

template <class T> - шаблон класса T

int l; - длина задаваемого вектора

T\* m; - указатель на первый элемент вектора

int StartIndex; - индекс первого элемента, для удобного индексирования элементов в верхнетреугольных матрицах.

**Класс Matrix:**

template<class A> - шаблон класса А

class Matrix : public Vector<Vector<A>> - наследование от класса Vector другого Vector, т.е. получается «вектор в векторе».

**Описание методов:**

|  |  |
| --- | --- |
| Метод: | Описание: |
| Vector<T>::Vector() | Конструктор по умолчанию для класса vector. |
| Vector<T>::Vector(const int \_l, const int \_si) | Конструктор vector, который принимает длину вектора и его стартовый индекс. |
| Vector<T>::Vector(const int \_l) | Конструктор vector, который принимает длину вектора. |
| Vector<T>::Vector(Vector<T> &A) | Конструктор копирования класса vector. |
| Vector<T>::~Vector() | Деструктор класса vector. |
| int Vector<T>::GetSize() | Метод, который возвращает размер текущего вектора. |
| int Vector<T>::GetStartIndex() | Метод, который возвращает стартовый индекс вектора. |
| T& Vector<T>::GetValue (const int n) | Метод, который позволяет получить значение конкретного элемента в векторе, по его индексу с его контролем. |
| T& Vector<T>::operator[]  (const int n) | Перегрузка оператора индексации. |
| Vector<T>& Vector<T>::operator=(const Vector<T>& A) | Перегрузка оператора присваивания векторов. |
| bool Vector<T>::operator==(const Vector<T>& A) | Перегрузка оператора равно, который возвращает «правда», если векторы равны, и «лож», если нет. |
| bool Vector<T>::operator!=(const Vector<T>& A) | Перегрузка оператора равно, который возвращает «лож», если векторы равны, и «правда», если нет. |
| Vector<T> Vector<T>::operator+(const Vector<T>& A) | Перегрузка оператора сложения векторов, к каждому элементу первого вектора прибавляется соответственный элемент второго. |
| Vector<T> Vector<T>::operator-(const Vector<T>& A) | Перегрузка оператора вычитания векторов, из каждого элемента первого вектора вычитается соответственный элемент второго. |
| T Vector<T>::operator\*(const Vector<T>& A) | Перегрузка оператора умножения векторов (скалярного), возвращается элемент типа T. |
| Vector<T> Vector<T>::operator+(const T& A) | Прибавление к каждому элементу вектора константной величины. |
| Vector<T> Vector<T>::operator-(const T& A) | Вычитание из каждого элемента вектора константной величины. |
| Vector<T> Vector<T>::operator\*(const T& A) | Умножение вектора на скаляр. |
| friend std::istream &operator>>(std::istream &in, Vector &a) | Перегрузка ввода вектора. |
| friend std::ostream &operator<<(std::ostream &out, Vector &a) | Перегрузка вывода вектора на консоль. |
| Matrix<A>::Matrix(int \_size) | Конструктор matrix, который принимает размер матрицы. |
| Matrix<A>::Matrix(Matrix <A> &B) : Vector<Vector<A>>(B) | Конструктор копирования матрицы, который наследуется от конструктора копирования вектора с шаблоном Vector<Vector<A>>(B). |
| Matrix<A>::Matrix(Vector<Vector<A>> &B) | Конструктор приведения типов из вектора «векторов» в матрицу. |
| Matrix<A>& Matrix<A>::operator=(Matrix<A>& B) | Перегрузка оператор присваивания для матриц. |
| bool Matrix<A>::operator==(Matrix<A>& B) | Перегрузка оператора равно для матриц. |
| Matrix<A> Matrix<A>::operator+(Matrix<A>& B) | Перегрузка оператора сложения для матриц. |
| Matrix<A> Matrix<A>::operator-(Matrix<A>& B) | Перегрузка оператора вычитания для матриц. |
| Matrix<A> Matrix<A>::operator\*(Matrix<A>& B) | Перегрузка оператора умножения для матриц. |
| Matrix<A> Matrix<A>::operator/(Matrix<A>& B) | Перегрузка оператора деления для матриц (умножение на обратную матрицу) |
| friend std::istream & operator>>(std::istream &in, Matrix &mt) | Перегрузка ввода матрицы. |
| friend std::ostream & operator<<(std::ostream &out, Matrix &mt) | Перегрузка вывода матрицы на консоль. |

## Описание алгоритмов

**Подробное описание некоторых методов**

Оператор присваивания вектора:

* Принимает константную ссылку на объект этого же класса, т.е. vector
* Если не идет попытка присвоить vector самому себе (if (m != A.m)), то поля вектора А присваиваются полям текущего вектора.
* Если в поле m присутствовали элементы, то массив очищается, и выделяется память нужного размера. После в цикле поэлементно присваиваются элементы вектора.
* В конце возвращается указатель на текущий объект.

Умножение матриц:

* Принимает ссылку на объект этого же класса, т.е. matrix.
* Создает матрицу размером l (результирующая матрица).
* В цикле от 0 до l (прохождение по строкам) идет цикл (столбцы матрицы) с i – го элемента (если отталкиваться от обычной матрицы, без представления «вектора векторов») до l, в котором еще 1 цикл от i до g + 1, в котором происходит перемножение элементов матрицы и их запись в результирующую матрицу.
* После всех циклов происходит возврат результирующей матрицы.

# Эксперименты

## Время выполнения

Рассмотрим время выполнения некоторых методов, различающихся свой теоретической сложностью (время приведено в наносекундах):

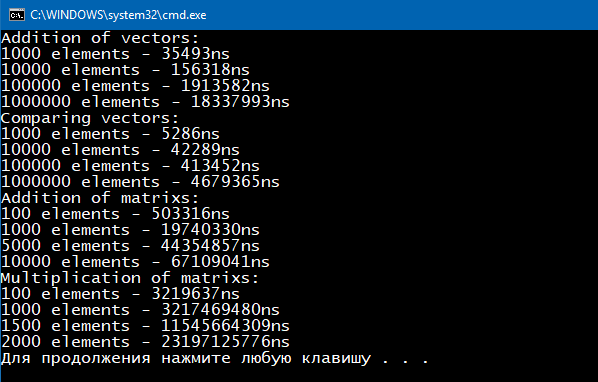
Сложение векторов – теоретическая сложность O(n), сравнение векторов – теоретическая сложность O(n), сложение матриц – теоретическая сложность O(n), умножение матриц – сложность O(n3). На практике получаются такие числа (рис. 2):

рис. 2 (время работы некоторых методов при разном количестве элементов)

Составим диаграмму с временем работы:

С учетом погрешности измерения, можно сказать, что теоретическая сложность алгоритмов совпадает с практическими измерениями.

Тесты проводились на системе:

Процессор Intel Core i5 7200U;

Оперативная память 12 GB.

# Заключение

В заключении можно сказать, что все поставленные цели и задачи были выполнены, а именно: созданы классы «vector» и «matrix» с реализованными методами сложения, вычитания и умножения матриц, а также написаны к ним тесты, и они успешно пройдены.

# Литература

* Учебные материалы к учебному курсу «Методы программирования» - Гергель В.П.

# Приложение

**vector.h:**

#pragma once

#include <iostream>

#include "Exception\_class.h"

template <class T>

class Vector

{

protected:

int l;

T\* m;

int tStartIndex;

Exceptions\_from\_vector\_and\_matrix except;

public:

Vector();

Vector(const int \_l, const int \_si);

Vector(const int \_l);

Vector(Vector<T> &a);

~Vector();

int GetSize();

int GetStartIndex();

T& GetValue(const int n);

T& operator[](const int n);

Vector<T>& operator=(const Vector<T>& a);

bool operator==(const Vector<T>& a);

bool operator!=(const Vector<T>& a);

Vector<T> operator+(const Vector<T>& a);

Vector<T> operator-(const Vector<T>& a);

T operator\*(const Vector<T>& a);

Vector<T> operator+(const T& a);

Vector<T> operator-(const T& a);

Vector<T> operator\*(const T& a);

friend std::istream &operator>>(std::istream &in, Vector &a)

{

for (int i = 0; i < a.l; i++)

in >> a.m[i];

return in;

}

friend std::ostream &operator<<(std::ostream &out, Vector &a)

{

for (int i = 0; i < a.l; i++)

out << a.m[i] << ' ';

return out;

}

};

template <class T>

Vector<T>::Vector()

{

l = 0;

m = NULL;

tStartIndex = 0;

}

template <class T>

Vector<T>::Vector(const int \_l, const int \_si)

{

if (\_l <= 0)

except.except\_throw(101);

tStartIndex = \_si;

l = \_l;

m = new T[l];

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

m[i] = 0;

}

template <class T>

Vector<T>::Vector(const int \_l)

{

if (\_l <= 0)

except.except\_throw(101);

l = \_l;

tStartIndex = 0;

m = new T[l];

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

m[i] = 0;

}

template <class T>

Vector<T>::Vector(Vector<T> &A)

{

tStartIndex = A.tStartIndex;

l = A.l;

m = new T[l];

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

m[i] = A.m[i];

}

template <class T>

Vector<T>::~Vector()

{

delete[] m;

}

template <class T>

int Vector<T>::GetSize()

{

return l;

}

template <class T>

int Vector<T>::GetStartIndex()

{

return tStartIndex;

}

template <class T>

T& Vector<T>::GetValue(const int n)

{

if (n < 0 || n > l - 1)

except.except\_throw(102);

return m[n];

}

template <class T>

T& Vector<T>::operator[](const int n)

{

if (n >= tStartIndex)

return m[n - tStartIndex];

except.except\_throw(102);

}

template <class T>

Vector<T>& Vector<T>::operator=(const Vector<T>& A)

{

if (m != A.m)

{

tStartIndex = A.tStartIndex;

l = A.l;

if (m)

delete[] m;

m = new T[l];

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

m[i] = A.m[i];

}

return \*this;

}

template <class T>

bool Vector<T>::operator==(const Vector<T>& A)

{

bool rez = true;

if (l != A.l)

rez = false;

else

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

if (m[i] != A.m[i])

rez = false;

return rez;

}

template <class T>

bool Vector<T>::operator!=(const Vector<T>& A)

{

bool rez = false;

if (l != A.l)

rez = true;

else

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

if (m[i] != A.m[i])

rez = true;

return rez;

}

template <class T>

Vector<T> Vector<T>::operator+(const Vector<T>& A)

{

if (l != A.l)

except.except\_throw(103);

int min = l;

if (A.l < l)

min = A.l;

Vector<T> rez(min, tStartIndex);

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

rez.m[i] = m[i] + A.m[i];

return rez;

}

template <class T>

Vector<T> Vector<T>::operator-(const Vector<T>& A)

{

if (l != A.l)

except.except\_throw(104);

int min = l;

if (A.l < l)

min = A.l;

Vector<T> rez(min, tStartIndex);

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

rez.m[i] = m[i] - A.m[i];

return rez;

}

template <class T>

T Vector<T>::operator\*(const Vector<T>& A)

{

if (l != A.l)

except.except\_throw(105);

T rez = 0;

if (A.l != l)

throw(1);

else

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

rez = rez + m[i] \* A.m[i];

return rez;

}

template <class T>

Vector<T> Vector<T>::operator+(const T& A)

{

Vector<T> rez(l, tStartIndex);

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

rez.m[i] = m[i] + A;

return rez;

}

template <class T>

Vector<T> Vector<T>::operator-(const T& A)

{

Vector<T> rez(l, tStartIndex);

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

rez.m[i] = m[i] - A;

return rez;

}

template <class T>

Vector<T> Vector<T>::operator\*(const T& A)

{

Vector<T> rez(l, tStartIndex);

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

rez.m[i] = m[i] \* A;

return rez;

}

**matrix.h:**

#pragma once

#include "Vector.h"

template<class A>

class Matrix : public Vector<Vector<A>>

{

public:

Matrix(int \_size);

Matrix(Matrix<A> &b);

Matrix(Vector<Vector<A>> &b);

Matrix<A>& operator=(Matrix<A>& b);

bool operator==(Matrix<A>& b);

Matrix<A> operator+(Matrix<A>& b);

Matrix<A> operator-(Matrix<A>& b);

Matrix<A> operator\*(Matrix<A>& b);

Matrix<A> operator/(Matrix<A>& b);

friend std::istream & operator>>(std::istream &in, Matrix &mt)

{

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

in >> mt.m[i];

return in;

}

friend std::ostream & operator<<(std::ostream &out, Matrix &mt)

{

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

{

if (mt.GetSize() > mt.m[i].GetSize())

{

int tmp = mt.GetSize() - mt.m[i].GetSize();

for (int j = 0; j < tmp; j++)

cout << 0 << " ";

}

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

}

return out;

}

};

template<class A>

Matrix<A>::Matrix(int \_size)

{

if (\_size < 0)

except.except\_throw(106);

else if (\_size == 0)

{

l = 0;

m = NULL;

tStartIndex = 0;

}

else

{

l = \_size;

m = new Vector<A>[l];

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

m[i] = Vector<A>(l - i, i);

}

}

template<class A>

Matrix<A>::Matrix(Matrix<A> &B) : Vector<Vector<A>>(B)

{

}

template<class A>

Matrix<A>::Matrix(Vector<Vector<A>> &B)

{

l = B.l;

tStartIndex = B.GetStartIndex;

m = new Vector<A>[l];

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

m[i] = B.m[i];

}

template<class A>

Matrix<A>& Matrix<A>::operator=(Matrix<A>& B)

{

if (m != B.m)

{

if (m)

{

delete[] m;

l = B.l;

tStartIndex = B.tStartIndex;

m = new Vector<A>[l];

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

m[i] = B.m[i];

}

}

return \*this;

}

template<class A>

bool Matrix<A>::operator==(Matrix<A>& B)

{

bool rez = true;

if (l != B.l)

rez = false;

else

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

if (m[i] != B.m[i])

rez = false;

return rez;

}

template<class A>

Matrix<A> Matrix<A>::operator+(Matrix<A>& B)

{

Matrix<A> rez(l);

if (l == B.l)

{

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

rez.m[i] = m[i] + B.m[i];

}

else

except.except\_throw(107);

return rez;

}

template<class A>

Matrix<A> Matrix<A>::operator-(Matrix<A>& B)

{

Matrix<A> rez(l);

if (l == B.l)

{

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

rez.m[i] = m[i] - B.m[i];

}

else

except.except\_throw(108);

return rez;

}

template<class A>

Matrix<A> Matrix<A>::operator\*(Matrix<A>& B)

{

if (l != B.l)

except.except\_throw(109);

Matrix<A> rez(l);

for (int i = 0; i < l; i++)//строки

for (int g = i; g < l; g++)//столбцы

for (int inner = i; inner < g + 1; inner++)

rez.m[i][g] += m[i][inner] \* B.m[inner][g];

return rez;

}

template<class A>

Matrix<A> Matrix<A>::operator/(Matrix<A>& B)

{

if (l != B.l)

except.except\_throw(110);

A det = B.m[0][0];

for (int i = 1; i < l; i++)

det = det \* B.m[i][i];

if (det == 0)

except.except\_throw(111);

Matrix <A> c1(B);

Matrix <A> res(l);

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

{

res[i][i] = 1;

A k = c1[i][i];

for (int j = i; j < l; j++)

{

c1[i][j] = c1[i][j] / k;

res[i][j] = res[i][j] / k;

}

}

for (int j = 1; j < l; j++)

for (int i = j - 1; i >= 0; i--)

{

A tmp = c1[i][j];

for (int k = j; k < l; k++)

{

res[i][k] = 0 - res[j][k] \* tmp;

c1[i][k] = c1[i][k] - c1[j][k] \* tmp;

}

}

return ((\*this) \* res);

}

**vector\_test.cpp:**

#include "Vector.h"

#include <gtest.h>

TEST(Vector, can\_create\_vector\_with\_positive\_lenght)

{

ASSERT\_NO\_THROW(Vector<int> bf(3));

}

TEST(Vector, can\_throw\_when\_create\_vector\_with\_zero\_lenght)

{

ASSERT\_ANY\_THROW(Vector<int> bf(0));

}

TEST(Vector, can\_return\_vector\_size)

{

Vector<int> bf(3);

EXPECT\_EQ(3, bf.GetSize());

}

TEST(Vector, can\_return\_element\_with\_negative\_or\_more\_sise\_index)

{

Vector<int> bf(3);

ASSERT\_ANY\_THROW(bf.GetValue(3));

}

TEST(Vector, can\_return\_element\_with\_normal\_index)

{

Vector<int> bf(3);

ASSERT\_NO\_THROW(bf.GetValue(2));

}

TEST(Vector, can\_return\_i\_element)

{

Vector<int> bf(3);

bf[1] = 4;

EXPECT\_EQ(4, bf[1]);

}

TEST(Vector, can\_assign\_vector)

{

Vector<int> bf(2), bp(2);

bf[0] = 5;

bp = bf;

EXPECT\_EQ(5, bp[0]);

}

TEST(Vector, can\_compare\_vectors)

{

Vector<int> bf(2), bp(2);

bf[1] = 5;

bf[0] = 3;

bp = bf;

EXPECT\_EQ(1, bf == bp);

}

TEST(Vector, can\_compare\_different\_vectors)

{

Vector<int> bf(2), bp(3);

bf[1] = 5;

bf[0] = 3;

EXPECT\_EQ(0, bf == bp);

}

TEST(Vector, can\_compare\_different\_vectors\_with\_not\_equal\_operator)

{

Vector<int> bf(2), bp(2);

bf[1] = 5;

bf[0] = 3;

bp[0] = 2;

EXPECT\_EQ(1, bf != bp);

}

TEST(Vector, can\_compare\_vectorswith\_different\_lenght\_with\_not\_equal\_operator)

{

Vector<int> bf(2), bp(3);

bf[1] = 5;

bf[0] = 3;

bp[0] = 3;

bp[1] = 5;

EXPECT\_EQ(1, bf != bp);

}

TEST(Vector, can\_add\_vectors)

{

Vector<int> bf(2), bp(2), bd(2);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

bd = bp + bf;

EXPECT\_EQ(2, bd[0]);

EXPECT\_EQ(4, bd[1]);

}

TEST(Vector, can\_throw\_if\_add\_vectors\_whis\_differnt\_size)

{

Vector<int> bf(2), bp(3);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

ASSERT\_ANY\_THROW(bp + bf);

}

TEST(Vector, can\_subtract\_vectors)

{

Vector<int> bf(2), bp(2), bd(2);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

bd = bp - bf;

EXPECT\_EQ(0, bd[0]);

EXPECT\_EQ(0, bd[1]);

}

TEST(Vector, can\_throw\_if\_subtract\_vectors\_whis\_differnt\_size)

{

Vector<int> bf(2), bp(3);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

ASSERT\_ANY\_THROW(bp + bf);

}

TEST(Vector, can\_conduct\_vector\_multiplication)

{

Vector<int> bf(2), bp(2);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

EXPECT\_EQ(5, bf \* bp);

}

TEST(Vector, can\_throw\_when\_vector\_multiplication\_with\_different\_lenght)

{

Vector<int> bf(2), bp(3);

for (int i = 0; i < 2; i++)

bp[i] = bf[i] = i + 1;

ASSERT\_ANY\_THROW(bf \* bp);

}

TEST(Vector, can\_add\_scalar\_to\_vector)

{

Vector<int> bf(2);

for (int i = 0; i < 2; i++)

bf[i] = i + 1;

bf = bf + 5;

EXPECT\_EQ(6, bf[0]);

EXPECT\_EQ(7, bf[1]);

}

TEST(Vector, can\_subtract\_scalar\_to\_vector)

{

Vector<int> bf(2);

for (int i = 0; i < 2; i++)

bf[i] = i + 1;

bf = bf - 5;

EXPECT\_EQ(-4, bf[0]);

EXPECT\_EQ(-3, bf[1]);

}

TEST(Vector, can\_multiply\_scalar\_to\_vector)

{

Vector<int> bf(2);

for (int i = 0; i < 2; i++)

bf[i] = i + 1;

bf = bf \* 5;

EXPECT\_EQ(5, bf[0]);

EXPECT\_EQ(10, bf[1]);

}

**matrix\_test.cpp:**

#include <gtest.h>

#include "Matrix.h"

TEST(Matrix, can\_create\_matrix\_with\_negative\_size)

{

ASSERT\_ANY\_THROW(Matrix<int> bf(-3));

}

TEST(Matrix, can\_create\_matrix\_with\_positive\_size)

{

ASSERT\_NO\_THROW(Matrix<int> bf(3));

}

TEST(Matrix, can\_assign\_matrix)

{

Matrix<int> bf(2), bp(2);

bf[0][0] = 1;

bf[0][1] = 2;

bf[1][1] = 3;

bp = bf;

EXPECT\_EQ(1, bp[0][0]);

EXPECT\_EQ(2, bp[0][1]);

EXPECT\_EQ(3, bp[1][1]);

}

TEST(Matrix, can\_compare\_matrixs)

{

Matrix<int> bf(2), bp(2);

bf[0][0] = 5;

bf[0][1] = 3;

bf[1][1] = 1;

bp = bf;

EXPECT\_EQ(1, bf == bp);

}

TEST(Matrix, can\_compare\_different\_matrixs)

{

Matrix<int> bf(2), bp(0);

bf[0][0] = 5;

bf[0][1] = 3;

bf[1][1] = 1;

EXPECT\_EQ(0, bf == bp);

}

TEST(Matrix, can\_add\_matrixs)

{

Matrix<int> bf(2), bp(2), bd(2);

bf[0][0] = 5;

bf[0][1] = 3;

bf[1][1] = 1;

bp = bf;

bd = bp + bf;

EXPECT\_EQ(10, bd[0][0]);

EXPECT\_EQ(6, bd[0][1]);

EXPECT\_EQ(2, bd[1][1]);

}

TEST(Matrix, can\_throw\_if\_add\_matrixs\_with\_different\_lenght)

{

Matrix<int> bf(2), bp(3);

ASSERT\_ANY\_THROW(bf + bp);

}

TEST(Matrix, can\_subtract\_matrixs)

{

Matrix<int> bf(2), bp(2), bd(2);

bf[0][0] = 5;

bf[0][1] = 3;

bf[1][1] = 1;

bp[0][0] = 3;

bp[0][1] = 2;

bp[1][1] = 1;

bd = bf - bp;

EXPECT\_EQ(2, bd[0][0]);

EXPECT\_EQ(1, bd[0][1]);

EXPECT\_EQ(0, bd[1][1]);

}

TEST(Matrix, can\_throw\_if\_subtract\_matrixs\_with\_different\_lenght)

{

Matrix<int> bf(2), bp(3);

ASSERT\_ANY\_THROW(bf - bp);

}

TEST(Matrix, can\_multiply\_matrixs)

{

Matrix<int> bf(2), bp(2), bd(2);

bf[0][0] = 5;

bf[0][1] = 3;

bf[1][1] = 1;

bp[0][0] = 3;

bp[0][1] = 2;

bp[1][1] = 1;

bd = bf \* bp;

EXPECT\_EQ(15, bd[0][0]);

EXPECT\_EQ(13, bd[0][1]);

EXPECT\_EQ(1, bd[1][1]);

}

TEST(Matrix, can\_throw\_if\_multiply\_matrixs\_with\_different\_lenght)

{

Matrix<int> bf(2), bp(3);

ASSERT\_ANY\_THROW(bf \* bp);

}

TEST(Matrix, can\_throw\_if\_division\_matrixs\_with\_different\_lenght)

{

Matrix<int> bf(2), bp(3);

ASSERT\_ANY\_THROW(bf / bp);

}

TEST(Matrix, can\_throw\_if\_division\_matrixs\_with\_zero\_determinant)

{

Matrix<int> bf(2), bp(2);

ASSERT\_ANY\_THROW(bf / bp);

}

TEST(Matrix, can\_division\_matrix)

{

Matrix<double> bf(3), bp(3);

Matrix<double> resbf(3), resbp(3), res(3);

bf[0][0] = 5;

bf[0][1] = 3;

bf[0][2] = 1;

bf[1][1] = 3;

bf[1][2] = 2;

bf[2][2] = 1;

bp[0][0] = 1;

bp[0][1] = 2;

bp[0][2] = 3;

bp[1][1] = -1;

bp[1][2] = -2;

bp[2][2] = 4;

resbp[0][0] = 1;

resbp[0][1] = 2;

resbp[0][2] = 0.25;

resbp[1][1] = -1;

resbp[1][2] = -0.5;

resbp[2][2] = 0.25;

resbf = bf / bp;

res = bf \* resbp;

EXPECT\_EQ(resbf[0][0], res[0][0]);

EXPECT\_EQ(resbf[0][1], res[0][1]);

EXPECT\_EQ(resbf[0][2], res[0][2]);

}