Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики и механики

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчёт по лабораторной работе

**Реализация классов для работы с матрицами и векторами**

Выполнила:

студентка ИИТММ гр. 381906-2

Тюлькина О.В.

Проверил:

программист каф. МОСТ

Усова М. А.

Нижний Новгород

2020г.

**Содержание**

[1. Введение 3](#_Toc61723668)

[2. Постановка задачи 4](#_Toc61723669)

[3. Руководство пользователя 5](#_Toc61723670)

[4. Руководство программиста 6](#_Toc61723671)

[4.1. Описание структуры программы 6](#_Toc61723672)

[4.2. Описание структур данных 7](#_Toc61723673)

[4.3. Описание алгоритмов 9](#_Toc61723674)

[5. Эксперименты 10](#_Toc61723675)

[6. Заключение 11](#_Toc61723676)

[7. Литература 12](#_Toc61723677)

[8. Приложение 13](#_Toc61723678)

1. **Введение**

На сегодняшний день математическое программирование – неотъемлемая составляющая всего программирования. Огромную роль в нем играют матрицы и векторы. Матрица – это объект, записываемый в виде прямоугольной таблицы, которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Векторами называются матрицы, состоящие из одной строки или одного столбца.

Чаще всего в математике работают с матрицами общего вида, но наряду с ними используются матрицы специального вида – треугольные, диагональные. Матрицы и вектора – важные математические объекты, поэтому нужно научиться работать с ними.

Мной были созданы классы для работы с векторами и матрицами. Также я произвела теоретическую и практическую оценку времени работы различных арифметических операций над матрицами.

**2. Постановка задачи**

Реализовать классы для работы с векторами и матрицами, используя шаблоны. Продемонстрировать их работу на примере.

Должны быть:

* конструкторы (по умолчанию, инициализатор, копирования);
* деструктор;
* доступ к защищенным полям;
* перегруженные операции: +, -, \*, =, ==, [];
* потоковый ввод и вывод;
* перегруженные операции +, -, \*, / должны быть реализованы для векторов (вектор +, -, \*, / вектор или скаляр), матриц (матрица +, -, \* матрица или скаляр);

Создание программных средств для эффективного хранения матриц верхнетреугольного вида, выполнение основных операций как сложение/вычитание умножение, копирование, сравнение. В процессе выполнения лабораторной работы требуется использовать систему контроля версий Git и фрэймворк для разработки автоматических тестов Google Test. Сделать выводы о проделанной работе.

**3. Руководство пользователя**

Запускаем программу:

1.Создать математические объекты типа Matrix или Vector, в зависимости от необходимости.

2.Расширить объекты при помощи доступа к защищенным полям(гетеры) или при помощи конструкторов

3.Выполнить необходимые математические операции, производимые с векторами и матрицами, применимые в алгебре (умножение, сложение, вычитание матриц и векторов при подходящем размере, умножение вектора на матрицу и наоборот при помощи матричного умножения), проверка работы перегруженных операций, конструкторов.

**4. Руководство программиста**

**4.1. Описание структуры программы**

Программа состоит из нескольких блоков:

ALL\_BUILD, gtest, ZERO\_CHECK - вспомогательные записи, ZERO\_CHECK перезапустит cmake. ALL\_BUILD - это цель, которая строит весь проект в активное решение.

В блоке vectorlib содержится шаблонный класс TVector, объявлены все методы и их определения класса векторов, и его наследник TMatrix, в котором объявлены все методы класса матриц и их определения, объявлены и реализованы функции перегрузки операторов матрично-векторного умножения.

В блоке vector содержится файл main с примером работы с классами класс Vector, Matrix.

Также есть блок test, в которых содержится набор тестов, с помощью которых мы проверяем корректную работу программы.

**4.2. Описание структур данных**

В программе определены два следующих шаблонных класса:

Class TVector,

Class TMatrix

Внутри класса TVector определены следующие protected поля:

• ValType\* pVector; – шаблонный указатель;

• Int Size - длина вектора;

• Int StartIndex– номер стартовой ячейки вектора;

Внутри класса TVector определен следующий набор public-методов (а также конструкторы и деструктор):

* TVector (int s = 10, int si = 0) – конструктор по умолчанию и инциализатор, принимает длину вектора и стартовый индекс;
* TVector (const TVector& v) – конструктор копирования, принимает на вход объект типа TVector, создает объект с теми же характеристиками что и переданный, и массивом того же содержимого;
* ~TVector() – деструктор, очищает выделенную конструкторами память;
* int GetSize () – метод, возвращающий длину вектора;
* int GetStartIndex () – метод возвращающий стартовый индекс вектора;
* void SetSize(int) – метод устанавливающий размер вектора в соответствии с переданным аргументом;
* void SetStartIndex(int)- метод устанавливающий стартовый индекс в соответствии с переданным аргументом;
* VaType\* GetVector () – метод возвращающий указатель на блок памяти с вектором;
* ValType& operator[](int pos) – перегрузка оператора индексации;
* ValType& operator [ ](int pos) const - константная перегрузка оператора индексации;
* bool operator == (сonst TVector&v) – перегрузка оператора равно;
* bool operator != (сonst TVector&v) – перегрузка оператора неравно;
* TVector& operator = (const TVector&v) – перегрузка оператора присваивания;
* TVector operator +(const ValType& val) – перегрузка оператора сложения для объектов вектор и скаляр. Математически определяется как вектор с координатами, каждая из которых увеличена на скаляр;
* TVector operator -(const ValType&val) - перегрузка оператора вычитания, по своей сути аналогична перегрузки сложения;
* TVector operator \*(const ValType&val) - перегрузка оператора умножения на скаляр;
* TVector operator +(const TVector&v) - перегрузка оператора сложения для двух векторов;
* TVector operator -(const TVector&v) - перегрузка оператора вычитания, для двух векторов;
* ValType operator \*(const TVector&v) - перегрузка оператора умножения, для двух векторов;
* friend ostream& operator <<(ostream& out, const TVector& rhs) – перегрузка оператора вывода, позволяет выводить вектора на экран;
* friend istream& operator >>(istream& in, const TVector& rhs) - перегрузка оператора ввода, позволяет вводить вектора уже заданного размера, либо задавать размер и вводить самостоятельно.

Шаблонный класс TMatrix, является public-наследником класса TVector, при этом параметр шаблона определен как TVector <ValType> поэтому никаких полей в этом классе не определено. Также почти 70% методов, он наследует от класса-родителя, поэтому внутри него, определены только специфичные для него методы:

* + TMatrix(int s=10) –конструктор по умолчанию, и конструктор инциализатор ;
  + TMatrix(const TMatrix&mt) – конструктор копирования;
  + TMatrix(TVector<TVector<ValType> >&mt) конструктор с параметром вектор с шаблоном вектор от шаблона;
  + bool operator == (const TMatrix&mt)const – перегрузка оператора равно;
  + bool operator != (const TMatrix&mt)const – перегрузка оператора неравно;
  + TMatrix& operator = (const TMatrix&mt) – перегрузка оператора присваивания;
  + TMatrix operator + (const TMatrix&mt) – перегрузка оператора суммы;
  + TMatrix operator - (const TMatrix&mt) - перегрузка оператора разности;
  + friend ostream& operator <<(ostream&out, const TMatrix&mt) – перегрузка оператора вывода на экран;
  + friend istream& operator >>(istream&in, TMatrix&mt) - перегрузка оператора ввода.

**4.3. Описание алгоритмов**

**Алгоритм суммирования или вычитания векторов:**

Сначала выполняется проверка на совпадение размерности. Для сложения векторов необходимо сложить соответствующие координаты этих векторов. Создается вектор для записи результата. По циклу соответствующие координаты складываются или вычитаются, в результате работы возвращается результирующий вектор.

Так как класс TMatrix - публичный наследник класса TVector, то для него нет смысла описывать арифметическую операцию сложения, так как эта операция наследуется от класса TVector.

**Алгоритм умножения матриц:**

Если чисто столбцов в первой матрице совпадает с числом строк во второй, то эти матрицы можно перемножить. На входе выполняется эта проверка. Создается матрица для записи результата. В цикле каждая строка первой матрицы почленно умножается с каждым столбцом второй матрицы. Возвращается результирующая матрица.

Алгоритмы нахождения произведения вектора на матрицу и матрицы на вектор аналогичны алгоритму умножения матриц.

**5. Эксперименты**

Проверим корректность выполнения нашей программы. Проверим свойства верхнетреугольных матриц.

Найдем определитель матрицы. Определитель верхнетреугольной матрицы равен произведению чисел, стоящих на главной диагонали. Проверим, корректны ли вычисления определителя в работе нашей программы.

Зададим две матрицы.



Рисунок 1: Ввод данных матриц.

Проверим результат работы умножения матриц.



Рисунок 2: Результат произведения матриц.

Следовательно, программа работает корректно.

**6. Заключение**

В результате проведения данной работы были реализованы объекты для работы с матрицами и векторами.

Были определены арифметические операции для работы с матрицами, реализованными в данной структуре, проведены эксперименты, подтверждающие корректную работу программы. Также работоспособность программы проверена на тестах.

**7. Литература**

1. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003.
2. Сайт wikipedia – Режим доступа: <https://ru.wikipedia.org/wiki/>
3. Сайт habr – Режим доступа: <https://habr.com/ru/>

**8. Приложение**

**MyVector.h**

#pragma once

#include <iostream>

using namespace std;

const int MAX\_VECTOR\_SIZE = 100000000;

template <class ValType>

class TVector

{

protected:

ValType\* pVector;

int Size;

int StartIndex;

public:

TVector(int s = 10, int si = 0);

TVector(const TVector& v);

~TVector();

ValType\* GetVector()const

{

return pVector;

}

int GetSize() const { return Size; }

int GetStartIndex()const { return StartIndex; } /

ValType& operator[](int pos);

const ValType& operator[](int pos)const;

bool operator==(const TVector& v)const;

bool operator!=(const TVector& v) const;

TVector& operator=(const TVector& v);

// скалярные операции

TVector operator+(const ValType& val);

TVector operator-(const ValType& val);

TVector operator\*(const ValType& val);

// векторные операции

TVector operator+(const TVector& v);

TVector operator-(const TVector& v);

ValType operator\*(const TVector& v);

// ввод-вывод

friend istream& operator >> (istream& in, TVector& v)

{

for (int i = 0; i < v.Size; i++)

in >> v.pVector[i];

return in;

}

friend ostream& operator<<(ostream& out, const TVector& v)

{

for (int i = 0; i < v.Size; i++)

out << v.pVector[i] << ' ';

return out;

}

};

template <class ValType>

TVector<ValType>::TVector(int s, int si)

{

if ((s < 0) || (s > MAX\_VECTOR\_SIZE) || (si < 0)) throw logic\_error("negative size");

pVector = new ValType[s];

for (int i = si; i < s; i++)

{

pVector[i] = 0;

}

Size = s;

StartIndex = si;

}

template <class ValType>

TVector<ValType>::TVector(const TVector<ValType>& v)

{

pVector = new ValType[v.Size];

for (int i = v.StartIndex; i < v.Size; i++)

{

pVector[i] = v.pVector[i];

}

Size = v.Size;

StartIndex = v.StartIndex;

}

template <class ValType>

TVector<ValType>::~TVector()

{

delete[]pVector;

}

template <class ValType>

ValType& TVector<ValType>::operator[](int pos)

{

if ((pos < StartIndex) || (pos < 0) || (pos > StartIndex + Size)) throw logic\_error ("negative index");

return pVector[pos - StartIndex];

}

template <class ValType>

const ValType& TVector<ValType>::operator[](int pos)const

{

if ((pos < StartIndex) || (pos < 0) || (pos > StartIndex + Size)) throw logic\_error("negative index");

return pVector[pos - StartIndex];

}

template <class ValType>

bool TVector<ValType>::operator==(const TVector& v) const

{

if (((\*this).GetSize() == v.GetSize()) && ((\*this).GetStartIndex() == v.GetStartIndex()))

{

for (int i = v.GetStartIndex(); i < v.GetSize(); i++)

{

if ((\*this).GetVector()[i] != v.GetVector()[i])

{

return false;

}

}

return true;

}

return false;

}

template <class ValType>

bool TVector<ValType>::operator!=(const TVector& v) const

{

return!(v == (\*this));

}

template <class ValType>

TVector<ValType>& TVector<ValType>::operator=(const TVector& v)

{

if (this == &v)

{

return (\*this);

}

delete[](\*this).pVector;

pVector = new ValType[v.GetSize()];

for (int i = v.GetStartIndex(); i < v.GetSize(); i++)

{

pVector[i] = v.GetVector()[i];

}

Size = v.GetSize();

StartIndex = v.GetStartIndex();

return (\*this);

}

template <class ValType>

TVector<ValType> TVector<ValType>::operator+(const ValType& val)

{

TVector<ValType> res((\*this));

for (int i = res.GetStartIndex(); i < res.GetSize(); i++)

{

res[i] += val;

}

return res;

}

template <class ValType>

TVector<ValType> TVector<ValType>::operator-(const ValType& val)

{

TVector<ValType> res((\*this));

for (int i = res.GetStartIndex(); i < res.GetSize(); i++)

{

res[i] -= val;

}

return res;

}

template <class ValType>

TVector<ValType> TVector<ValType>::operator\*(const ValType& val)

{

TVector<ValType> res((\*this));

for (int i = res.GetStartIndex(); i < res.GetSize(); i++)

{

res[i] \*= val;

}

return res;

}

template <class ValType>

TVector<ValType> TVector<ValType>::operator+(const TVector<ValType>& v)

{

if ((\*this).GetSize() != v.GetSize()) throw logic\_error("not equal size");

TVector<ValType> res((\*this));

for (int i = res.GetStartIndex(); i < res.GetSize(); i++)

{

res[i] += v.GetVector()[i];

}

return res;

}

template <class ValType>

TVector<ValType> TVector<ValType>::operator-(const TVector<ValType>& v)

{

if ((\*this).GetSize() != v.GetSize()) throw logic\_error("not equal size");

TVector<ValType> res((\*this));

for (int i = res.GetStartIndex(); i < res.GetSize(); i++)

{

res[i] -= v.GetVector()[i];

}

return res;

}

template <class ValType>

ValType TVector<ValType>::operator\*(const TVector<ValType>& v)

{

if ((\*this).GetSize() != v.GetSize()) throw logic\_error("not equal size");

ValType res = 0;

for (int i = v.GetStartIndex(); i < v.GetSize(); i++)

{

res += v.GetVector()[i] \* (\*this).GetVector()[i];

}

return sqrt(res);

}

**MyMatrix.h**

#pragma once

#include "MyVector.h"

const int MAX\_MATRIX\_SIZE = 10000;

template <class ValType>

class TMatrix : public TVector<TVector<ValType> >

{

public:

TMatrix(int s = 10);

TMatrix(const TMatrix& mt);

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

bool operator==(const TMatrix& mt) const;

bool operator!=(const TMatrix& mt) const;

TMatrix& operator= (const TMatrix& mt);

TMatrix operator+ (const TMatrix& mt);

TMatrix operator- (const TMatrix& mt);

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

{

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

in >> mt.pVector[i];

return in;

}

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

{

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

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

return out;

}

};

template <class ValType>

TMatrix<ValType>::TMatrix(int s) : TVector<TVector<ValType> >(s)

{

if ((s < 0) || (s > MAX\_MATRIX\_SIZE)) throw logic\_error("incorrect parametr");

for (int i = 0; i < this->Size - this->StartIndex; i++)

{

TVector<ValType>res(s - i, i);

this->pVector[i] = res;

}

}

template <class ValType>

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

{

for (int i = this->StartIndex; i < this->Size; i++)

{

TVector<ValType>res(mt[i]);

(\*this)[i] = res;

}

Size = mt.GetSize();

StartIndex = mt.GetStartIndex();

}

template <class ValType>

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

{

for (int i = (\*this).GetStartIndex(); i < (\*this).GetSize(); i++)

{

TVector<ValType>ms(mt[i]);

(\*this)[i] = ms;

}

Size = mt.GetSize();

StartIndex = mt.GetStartIndex();

}

template <class ValType>

bool TMatrix<ValType>::operator==(const TMatrix<ValType>& mt) const

{

if (((\*this).GetStartIndex() == mt.GetStartIndex()) && ((\*this).GetSize() == mt.GetSize()))

{

for (int i = mt.GetStartIndex(); i < mt.GetSize(); i++)

{

if ((\*this).GetVector()[i] != mt.GetVector()[i])

{

return false;

}

}

return true;

}

return false;

}

template <class ValType>

bool TMatrix<ValType>::operator!=(const TMatrix<ValType>& mt) const

{

return !((\*this) == mt);

}

template <class ValType>

TMatrix<ValType>& TMatrix<ValType>::operator=(const TMatrix<ValType>& mt)

{

if (&mt == this)

{

return (\*this);

}

delete[]pVector;

pVector = new TVector<ValType>[mt.GetSize()];

(\*this).Size = mt.GetSize();

(\*this).StartIndex = mt.GetStartIndex();

for (int i = (\*this).GetStartIndex(); i < (\*this).GetSize(); i++)

{

(\*this)[i] = mt.GetVector()[i];

}

return (\*this);

}

template <class ValType>

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

{

if (mt.GetSize() != (\*this).GetSize()) throw logic\_error("incorrect size");

TMatrix<ValType>Res(\*this);

for (int i = Res.GetStartIndex(); i < Res.GetSize(); i++)

{

Res[i] = (\*this).GetVector()[i] + mt.GetVector()[i];

}

return Res;

}

template <class ValType>

TMatrix<ValType> TMatrix<ValType>::operator-(const TMatrix<ValType>& mt)

{

if (mt.GetSize() != (\*this).GetSize()) throw logic\_error("incorrect size");

TMatrix<ValType>Res(\*this);

for (int i = Res.GetStartIndex(); i < Res.GetSize(); i++)

{

Res[i] = (\*this)[i] - mt.GetVector()[i];

}

return Res;

}

**Main.cpp**

#include <iostream>

#include "MyMatrix.h"

int main()

{

TMatrix<int> a(5), b(5), c(5);

int i, j;

for (i = 0; i < 5; i++)

for (j = i; j < (i + 1); j++)

{

a[i][j] = i \* 10 + j;

b[i][j] = (i \* 10 + j) \* 100;

}

c = b + a;

cout << "Matrix a = " << endl << a << endl;

cout << "Matrix b = " << endl << b << endl;

cout << "Matrix c = a + b" << endl << c << endl;

return 0;

}

**test\_matrix.cpp**

#include "MyMatrix.h"

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

TEST(TMatrix, can\_create\_matrix\_with\_positive\_length)

{

ASSERT\_NO\_THROW(TMatrix<int> m(5));

}

TEST(TMatrix, cant\_create\_too\_large\_matrix)

{

ASSERT\_ANY\_THROW(TMatrix<int> m(MAX\_MATRIX\_SIZE + 1));

}

TEST(TMatrix, can\_create\_copied\_matrix)

{

TMatrix<int> m(5);

ASSERT\_NO\_THROW(TMatrix<int> m1(m));

}

TEST(TMatrix, copied\_matrix\_is\_equal\_to\_source\_one)

{

TMatrix<int> m(5);

TMatrix<int>m1(m);

EXPECT\_EQ(m, m1);

}

TEST(TMatrix, copied\_matrix\_has\_its\_own\_memory)

{

TMatrix<int>A(5);

TMatrix<int>B(A);

EXPECT\_NE(A.GetVector(), B.GetVector());

}

TEST(TMatrix, can\_get\_size)

{

TMatrix<int>Res(10);

EXPECT\_EQ(Res.GetSize(), 10);

}

TEST(TMatrix, can\_set\_and\_get\_element)

{

TMatrix<int>Res(10);

Res[1][0] = 2;

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

}

TEST(TMatrix, throws\_when\_set\_element\_with\_negative\_index)

{

TMatrix<int>res(1);

ASSERT\_ANY\_THROW(res[-1][-2]);

}

TEST(TMatrix, throws\_when\_set\_element\_with\_too\_large\_index)

{

TMatrix<int>res(MAX\_MATRIX\_SIZE);

ASSERT\_ANY\_THROW(res[MAX\_MATRIX\_SIZE + 3]);

}

TEST(TMatrix, assign\_operator\_change\_matrix\_size)

{

TMatrix<int>A(10);

TMatrix<int>B(1);

B = A;

EXPECT\_EQ(B.GetSize(), 10);

}

TEST(TMatrix, compare\_matrix\_with\_itself\_return\_true)

{

TMatrix<int>A(10);

EXPECT\_EQ(A, A);

}

TEST(TMatrix, matrices\_with\_different\_size\_are\_not\_equal)

{

TMatrix<int>A(10);

TMatrix<int>B(1);

EXPECT\_NE(A, B);

}

TEST(TMatrix, can\_add\_matrices\_with\_equal\_size)

{

TMatrix<int>A(10);

TMatrix<int>B(10);

ASSERT\_NO\_THROW(A + B);

}

TEST(TMatrix, cant\_add\_matrices\_with\_not\_equal\_size)

{

TMatrix<int>A(10);

TMatrix<int>B(11);

ASSERT\_ANY\_THROW(A + B);

}

TEST(TMatrix, cant\_subtract\_matrixes\_with\_not\_equal\_size)

{

TMatrix<int>A(10);

TMatrix<int>B(20);

ASSERT\_ANY\_THROW(A - B);

}

**test\_vector.cpp**

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

#include "MyVector.h"

TEST(TVector, can\_create\_vector\_with\_positive\_length)

{

ASSERT\_NO\_THROW(TVector<int> v(5));

}

TEST(TVector, cant\_create\_too\_large\_vector)

{

ASSERT\_ANY\_THROW(TVector<int> v(MAX\_VECTOR\_SIZE + 1));

}

TEST(TVector, can\_create\_vector\_with\_max\_size)

{

ASSERT\_NO\_THROW(TVector<int> v(MAX\_VECTOR\_SIZE));

}

TEST(TVector, can\_create\_vector\_with\_zero\_size)

{

ASSERT\_NO\_THROW(TVector<int> v(0));

}

TEST(TVector, throws\_when\_create\_vector\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TVector<int> v(-5));

}

TEST(TVector, throws\_when\_create\_vector\_with\_negative\_startindex)

{

ASSERT\_ANY\_THROW(TVector<int> v(5, -2));

}

TEST(TVector, can\_create\_copied\_vector)

{

TVector<int> v(10);

ASSERT\_NO\_THROW(TVector<int> v1(v));

}

TEST(TVector, copied\_vector\_is\_equal\_to\_source\_one)

{

TVector<int>v1(10);

TVector<int>v2(v1);

EXPECT\_EQ(v1, v2);

}

TEST(TVector, copied\_vector\_has\_its\_own\_memory)

{

TVector<int>A(5);

TVector<int>B(A);

EXPECT\_NE(A.GetVector(), B.GetVector());

}

TEST(TVector, can\_set\_and\_get\_element)

{

TVector<int> v(4);

v[0] = 4;

EXPECT\_EQ(4, v[0]);

}

TEST(TVector, throws\_when\_set\_element\_with\_negative\_index)

{

TVector<int>P(2);

ASSERT\_ANY\_THROW(P[-1]);

}

TEST(TVector, vectors\_with\_different\_size\_are\_not\_equal)

{

TVector<int>A(10);

TVector<int>B(12);

EXPECT\_NE(A, B);

}

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

{

TVector<int>A(3), B(3);

A = A + 3;

for (int i = B.GetStartIndex(); i < B.GetSize(); i++)

{

B[i] = 3;

}

EXPECT\_EQ(A, B);

}

TEST(TVector, cant\_add\_vectors\_with\_not\_equal\_size)

{

TVector<int>A(3), B(4);

ASSERT\_ANY\_THROW(A + B);

}

TEST(TVector, cant\_multiply\_vectors\_with\_not\_equal\_size)

{

TVector<int>A(4), B(5);

ASSERT\_ANY\_THROW(A\*B);

}