МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

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

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА №2**

**«Структура хранения для матриц специального вида»**

**Выполнил:** студент группы 381903-3

Пасека Иван Евгеньевич

**Руководитель:**

Ассистент кафедры МОСТ

Лебедев Илья Геннадьевич

Нижний Новгород

2020

Содержание

[1.Введение 3](#_Toc532897542)

[2. Цели и задачи 4](#_Toc532897543)

[2.1. Используемые инструменты 4](#_Toc532897544)

[3. Руководство пользователя 5](#_Toc532897545)

[4. Руководство программиста 7](#_Toc532897546)

[4.1. Описание структуры программы 7](#_Toc532897547)

[4.2. Описание функций и процедур, их алгоритмов 7](#_Toc532897548)

[5. Эксперименты 11](#_Toc532897549)

[6. Заключение 12](#_Toc532897550)

[7. Литература 13](#_Toc532897551)

[8. Приложения 14](#_Toc532897552)

[8.1. Приложение 1:Класс TVector 14](#_Toc532897553)

[8.2. Приложение 2:Класс TMatrix 17](#_Toc532897554)

[8.3. Приложение 3:Код программы тестирования и экспериментов 19](#_Toc532897555)

[8.4. Приложение 4:Тесты для классов 21](#_Toc532897556)

# Введение

Матричные обозначения широко распространены в современной математике и её приложениях. *Матрица* – полезный аппарат для исследования многих задач теоретической и прикладной математики. Так, одной из важнейших является задача нахождения решения систем линейных алгебраических уравнений.

Следствием разнообразия областей применения матричного аппарата в современной науке является наличие в любом из больших математических программных комплексов (Mathcad, Mathematica, Derive, Mapple) подсистем, выполняющих операции над матрицами, а также существование специальных программных библиотек (ScalaPack, PlaPack), рассчитанных на обработку огромных (десятки и сотни тысяч строк) матриц, в том числе с использованием распределенных (параллельных) вычислений.

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, …). Для таких матриц предпочтительно создание собственных способов хранения и обработки, учитывающих специфику их структуры, и потому более эффективных. Изучению некоторых из них посвящена данная работа.

# 2. Цели и задачи

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих эффективное хранение матриц специального вида (верхнетреугольных) и выполнение основных операций над ними:

* сложение/вычитание;
* копирование;
* сравнение.

В процессе выполнения лабораторной работы требуется использовать систему контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Выполнение работы предполагает решение следующих задач:

1. Реализация методов шаблонного класса TVector согласно заданному интерфейсу.
2. Реализация методов шаблонного класса TMatrix согласно заданному интерфейсу.
3. Обеспечение работоспособности тестов и примера использования.
4. Реализация заготовок тестов, покрывающих все методы классов TVector и TMatrix.
5. Модификация примера использования в тестовое приложение, позволяющее задавать матрицы и осуществлять основные операции над ними.

## 2.1. Используемые инструменты

* Система контроля версий [Git](https://git-scm.com/book/ru/v2). Рекомендуется использовать один из следующих клиентов на выбор студента:
  + [Git](https://git-scm.com/downloads)
  + [GitHub Desktop](https://desktop.github.com/)
* Фреймворк для написания автоматических тестов [Google Test](https://github.com/google/googletest).
* Среда разработки Microsoft Visual Studio (2008 или старше).
* Опционально. Утилита [CMake](http://www.cmake.org/) для генерации проектов по сборке исходных кодов. Может быть использована для генерации решения для среды разработки, отличной от Microsoft Visual Studio 2008 или 2010.

# 3. Руководство пользователя

Запускаем программу из файла sample\_matrix.cpp.

Программа выведет пример двух верхнетреугольных матриц, результат их сложения, а потом пример перемножения двух матриц. (Рис.1):

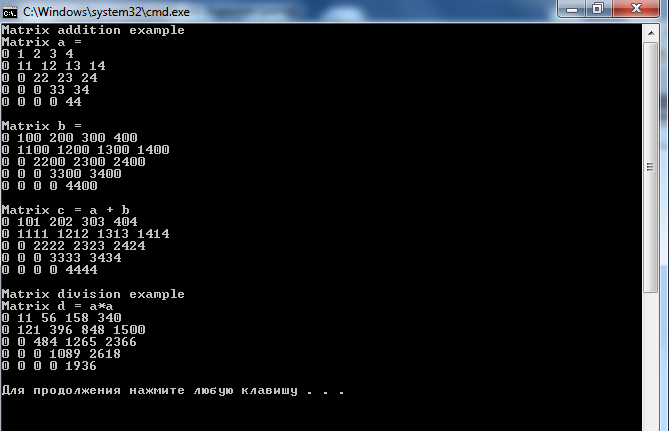


Рис.1. Сложение и умножение матриц.

Запускаем программу из файла sample\_input\_output.cpp (Рис.2):

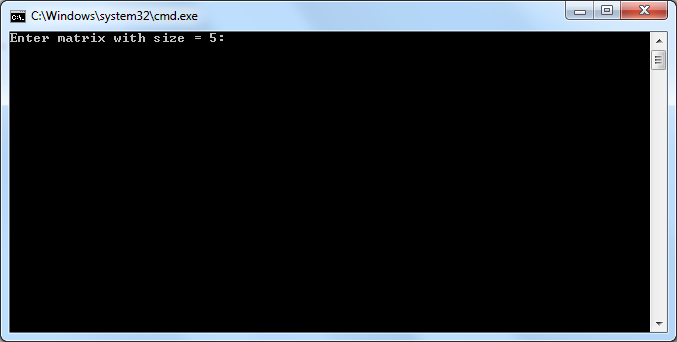


Рис.2.Начало.

В появившемся окне консоли пользователю необходимо ввести верхнетреугольную матрицу размера 5x5, например: (Рис.3).

1 2 3 4 5 press Enter

4 3 2 1 press Enter

1 2 3 press Enter

2 1 press Enter

1 press Enter

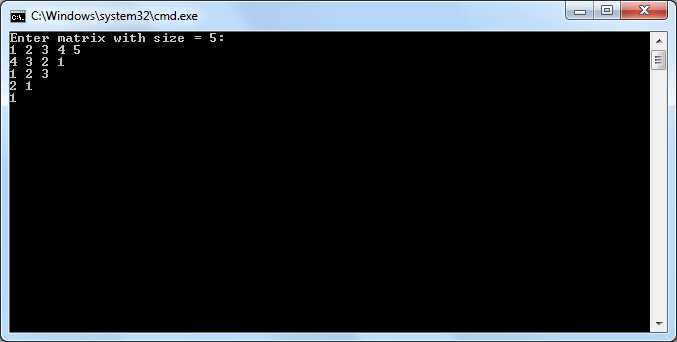


Рис.3.Ввод матрицы.

Программа выведет заданную матрицу в следующем виде: (Рис.4):

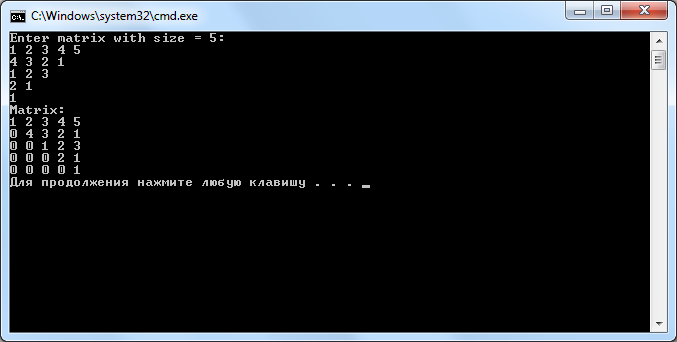


Рис.4.Вывод матрицы.

Алгоритм можно повторить несколько раз, используя другие значения.

# 4. Руководство программиста

Разработка системы вычисления проводились в среде “Microsoft Visual Studio 2010”.

Для реализации алгоритмов будет использовано 2 шаблонных класса:

* Класс «Вектор» (TVector), реализованный с использованием массива
* Класс «Матрица» (TMatrix), реализованный с использованием класса вектора.

## 4.1. Описание структуры программы

Модульная структура программы:

1. utvector.h, utvector.cpp – модуль с классом TVector, в котором определен интерфейс шаблонного класса и реализация его методов.
2. utmatrix.h, utmatrix.cpp – модуль с классом TMatrix, использующим класс TVector, в котором определен интерфейс шаблонного класса и реализация его методов.

* sample\_matrix.cpp , sample\_input\_output.cpp, sample\_performance\_check.cpp– модуль программы тестирования, с которым работает пользователь, в котором проводятся эксперименты.

1. test\_main.cpp, test\_tvector.cpp, test\_tmatrix.cpp – модуль с функциями тестирования для созданных классов. Содержат 28 тестов для класса TVector и 25 теста для класса TMatrix.

## 4.2. Описание функций и процедур, их алгоритмов

Рассмотрим реализацию методов шаблонного класса TVector:

template <class VectorType>

class TVector

1. TVector(int s = 1, int si = 0) - конструктор класса, принимающий длину вектора и стартовый индекс элемента. По умолчанию создается вектор длины 1 со стартовым индексом элемента 0.

2. TVector(const TVector &v) - конструктор копирования.

Принимает ссылку на объект класса TVector.

3. ~TVector() - деструктор. Освобождает выделенную под вектор память.

4. int GetSize() - возвращает длину вектора.

5. int GetStartIndex() - возвращает стартовый индекс элемента.

6. VectorType& operator[](int pos) – перегрузка оператора индексации.

Возвращает элемент, стоящий на заданной позиции.

7. bool operator == (const TVector &v) const – перегрузка оператора сравнения.

Выполняется проверка векторов на равенство. Возвращает true, если равенство выполняется, false в противном случае.

8. bool operator!=(const TVector &v) const - перегрузка оператора сравнения.

Выполняется проверка векторов на неравенство. Возвращает true, если неравенство выполняется, false в противном случае.

9. TVector& operator=(const TVector &v) – перегрузка оператора присваивания одного вектора другому.

Присваивает полям первого объекта класса поля второго объекта класса.

10. TVector operator+(const VectorType &val) – перегрузка оператора сложения элемента вектора с числом.

Возвращает вектор, равный поэлементного сложения исходного вектора с числом.

11. TVector operator-(const VectorType &val) - перегрузка оператора вычитания числа из элемента вектора.

Возвращает вектор, равный поэлементной разности исходного вектора с числом.

12. TVector operator\*(const VectorType &val) – умножение вектора на число.

Возвращает вектор, равный поэлементному произведению исходного вектора на число.

13. TVector operator+(const TVector &v) – перегрузка оператора сложения векторов.

Если длины векторов и стартовые индексы элементов совпадают, то к первому вектору поэлементного прибавляется второй.

14. TVector operator-(const TVector &v) - перегрузка оператора вычитания векторов.

Если длины векторов и стартовые индексы элементов совпадают, то из одного вектора поэлементного вычитается другой.

15. VectorType operator\*(const TVector &v) – перегрузка оператора скалярного произведения.

Если длины векторов и стартовые индексы элементов совпадают, то создается временная переменная, в которую записывается сумма произведений элементов соответствующих векторов.

Дружественные функции класса TVector:

friend istream& operator>>(istream &in, TVector &v) - ввод вектора через консоль. Принимает ссылку на стандартный поток ввода и ссылку на объект класса TVector, возвращает ссылку на стандартный поток ввода.

friend ostream& operator<<(ostream &out, const TVector &v) - вывод вектора на консоль.

Принимает ссылку на стандартный поток вывода и ссылку на объект класса TVector, возвращает ссылку на стандартный поток вывода.

Рассмотрим реализацию методов шаблонного класса TMatrix:

template <typename MatrixType>

class TMatrix : public TVector<TVector<MatrixType>>

1. TMatrix(int s = 10) - конструктор класса, принимающий размер матрицы (количество векторов).

По умолчанию создается матрица 10х10.

2. TMatrix(const TMatrix &mt) - конструктор копирования.

Принимает ссылку на объект класса TMatrix.

3. TMatrix(const TVector<TVector<MatrixType> > &mt) - конструктор преобразования типа.

4. bool operator==(const TMatrix &mt) const– перегрузка оператора сравнения.

Вызывается аналогичный оператор сравнения для TVector и возвращается результат его работы.

5. bool operator!=(const TMatrix &mt) const – перегрузка оператора сравнения.

Вызывается аналогичный оператор сравнения для TVector и возвращается результат его работы.

6. TMatrix& operator= (const TMatrix &mt) – перегрузка оператора присваивания.

Вызывается оператор присваивания для TVector. Возвращается ссылка на объект класса TMatrix.

7. TMatrix operator+ (const TMatrix &mt) – перегрузка оператора сложения двух матриц.

Вызывается оператор сложения для TVector и возвращается результат его работы.

8. TMatrix operator- (const TMatrix &mt) - перегрузка оператора вычитания одной матрицы из другой.

Вызывается оператор разности для TVector и возвращается результат его работы.

9. TMatrix inverse\_matrix() const – метод класса, который возвращает обратную матрицу.

10. TMatrix operator\* (const TMatrix &mt) – перегрузка оператора умножения для двух матриц.

Реализация перегрузки оператора умножения представлена, с помощью трех вложенных циклов:

11. TMatrix operator/ (const TMatrix &mt) – перегрузка оператора деления для двух матриц.

Использует метод inverse\_matrix() для приходящей матрицы, получая обратную ей. Перемножает две матрицы.

Дружественные функции класса TMatrix:

friend istream& operator>>(istream &in, TMatrix &mt) - ввод матрицы через консоль. Принимает ссылку на стандартный поток ввода и ссылку на объект класса TMatrix, возвращает ссылку на стандартный поток ввода.

friend ostream & operator<<( ostream &out, const TMatrix &mt) - вывод матрицы на консоль. Принимает ссылку на стандартный поток вывода и ссылку на объект класса TMatrix, возвращает ссылку на стандартный поток вывода.

# 5. Эксперименты

В качестве примера рассмотрим перегрузку оператора сложения для класса матрицы (TMatrix).

Теоретическая сложность выполнения алгоритма O(n2).

Мы провели измерение присваивая размер 10, 100, …, 1000000 матрице. Ниже вы можете увидеть график зависимости времени выполнения операции сложения от размера матрицы. (Рис.5) По приведенным данным можно сделать вывод, что практическая сложность выполнения алгоритма равна теоретической.

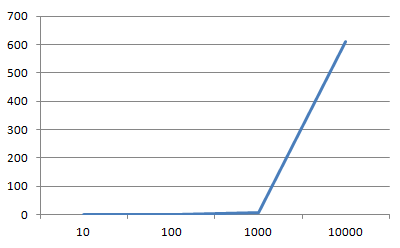


Рис.4. График зависимости времени выполнения операции сложения от размера матрицы.

По горизонтали – размер матрицы.

По вертикали - время выполнения программы.

# 6. Заключение

В результате лабораторной работы была разработана структура данных для хранения вектора и верхнетреугольной матрицы, а также освоены такие инструменты разработки программного обеспечения, как система контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Созданные шаблонные классы были протестированы с использованием Google Tests, а также были проведены эксперименты для сравнения теоретической и практической сложности выполнения операций на методе класса.

# 7. Литература

1. Лабораторный практикум: Учебно-методическое пособие / Мееров И.Б. [и др.] – Нижний Новгород: Нижегородский госуниверситет , 2017. – 105с.
2. Тестирование с использованием Google Test

(http://neerc.ifmo.ru/wiki/index.php?title=Тестирование\_с\_использованием\_Google\_Test#.D0.A4.D1.83.D0.BD.D0.BA.D1.86.D0.B8.D1.8F\_main.28.29)

1. Касперски К. Техника оптимизации программ. Эффективное использование памяти. — СПб.: БХВ-Петербург, 2003. — 464 с.: ил.
2. Основные принципы ООП: : инкапсуляция, наследование, полиморфизм, наследование.

(http://gos-it.wikia.com/wiki/Основные\_принципы\_ООП:\_инкапсуляция,\_наследование,\_полиморфизм)

1. https://ru.wikipedia.org/wiki/Треугольная\_матрица

# 8. Приложения

## 8.1. Приложение 1:Класс TVector

|  |
| --- |
| **utvector.h** |
| #pragma once  #include <iostream>  using namespace std;  #define MAX\_VECTOR\_SIZE 1000000  template <class VectorType>  class TVector  {  protected:  VectorType \*pVector;  int size;  int startIndex;  public:  TVector(int s = 1, int si = 0);  TVector(const TVector &v);  ~TVector();  int GetSize() { return size; }  int GetStartIndex(){ return startIndex; }  VectorType& operator[](int pos);  bool operator==(const TVector &v) const;  bool operator!=(const TVector &v) const;  TVector& operator=(const TVector &v);  TVector operator+(const VectorType &val);  TVector operator-(const VectorType &val);  TVector operator\*(const VectorType &val);  TVector operator+(const TVector &v);  TVector operator-(const TVector &v);  VectorType operator\*(const TVector &v);  // input\output  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;  }  }; |

|  |
| --- |
| **utvector.cpp** |
| #include "utvector.h"  template <class VectorType>  TVector<VectorType>::TVector(int s, int si):size(s),startIndex(si)  {  if ( (s > MAX\_VECTOR\_SIZE)||(s <= 0) )  throw 1;  else if ( (si > MAX\_VECTOR\_SIZE)||(si < 0) )  throw 1;  else  pVector = new VectorType [s];  }  template <class VectorType>  TVector<VectorType>::TVector(const TVector<VectorType> &v)  {  startIndex=v.startIndex;  size=v.size;  pVector = new VectorType[size];  for ( int i = 0; i < size; i++ )  pVector[i]=v.pVector[i];  }  template <class VectorType>  TVector<VectorType>::~TVector()  {  delete [] pVector;  }  template <class VectorType>  VectorType& TVector<VectorType>::operator[](int pos)  {  if ( pos < startIndex || pos >= startIndex + size )  throw 3;  return pVector[pos-startIndex];  }  template <class VectorType>  bool TVector<VectorType>::operator==(const TVector &v) const  {  if ( size!=v.size || startIndex!=v.startIndex )  return false;  for ( int i = 0; i < size; i++ )  {  if ( pVector[i] != v.pVector[i] )  return false;  }  return true;  }  template <class VectorType>  bool TVector<VectorType>::operator!=(const TVector &v) const  {  return !TVector<VectorType>::operator==(v);  }  template <class VectorType>  TVector<VectorType>& TVector<VectorType>::operator=(const TVector &v)  {  if( &v != this )  {  startIndex = v.startIndex;  if ( size != v.size )  {  size=v.size;  delete [] pVector;  pVector = new VectorType [size];  }  for ( int i = 0; i < size; i++ )  pVector[i]=v.pVector[i];  }  return \*this;  }  template <class VectorType>  TVector<VectorType> TVector<VectorType>::operator+(const VectorType &val)  {  for ( int i = 0; i < size; i++ )  pVector[i] += val;  return \*this;  }  template <class VectorType>  TVector<VectorType> TVector<VectorType>::operator-(const VectorType &val)  {  for ( int i = 0; i < size; i++ )  pVector[i] -= val;  return \*this;  }  template <class VectorType>  TVector<VectorType> TVector<VectorType>::operator\*(const VectorType &val)  {  for ( int i = 0; i < size; i++ )  pVector[i] \*= val;  return \*this;  }  template <class VectorType>  TVector<VectorType> TVector<VectorType>::operator+(const TVector<VectorType> &v)  {  if ( size != v.size || startIndex != v.startIndex )  throw 4;  else  {  for ( int i = 0; i < size; i++ )  pVector[i] = pVector[i] + v.pVector[i];  }  return \*this;  }  template <class VectorType>  TVector<VectorType> TVector<VectorType>::operator-(const TVector<VectorType> &v)  {  if ( size != v.size || startIndex != v.startIndex )  throw 4;  else  {  for ( int i = 0; i < size; i++ )  pVector[i] = pVector[i] - v.pVector[i];  }  return \*this;  }  template <class VectorType>  VectorType TVector<VectorType>::operator\*(const TVector<VectorType> &v)  {  VectorType result=0;  if ( size != v.size || startIndex != v.startIndex )  throw 4;  else  {  for ( int i = 0; i < size; i++ )  result += pVector[i] \* v.pVector[i];  }  return result;  } |

## 8.2. Приложение 2:Класс TMatrix

|  |
| --- |
| **utmatrix.h** |
| #pragma once  #include <iostream>  #include "utvector.h"  using namespace std;  #define MAX\_MATRIX\_SIZE 10000  template <typename MatrixType>  class TMatrix : public TVector<TVector<MatrixType> >  {  TMatrix inverse\_matrix() const;  public:  TMatrix(int s = 10);  TMatrix(const TMatrix &mt);  TMatrix(const TVector<TVector<MatrixType> > &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);  TMatrix operator\* (const TMatrix &mt);  TMatrix operator/ (const TMatrix &mt);  // input\output  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++ ) {  for ( int j = 0; j< i; j++ )  out << "0 ";  out << mt.pVector[i] << endl;  }  return out;  }  }; |

|  |
| --- |
| **utmatrix.cpp** |
| #include "utmatrix.h"  template <class MatrixType>  TMatrix<MatrixType>::TMatrix(int s): TVector<TVector<MatrixType> >(s)  {  if ( s > MAX\_MATRIX\_SIZE )  throw 5;  for ( int i = 0; i < s; i++ )  this -> pVector[i] = TVector<MatrixType>(s - i, i);  }  template <class MatrixType>  TMatrix<MatrixType>::TMatrix(const TMatrix<MatrixType> &mt):  TVector<TVector<MatrixType> >(mt) {}  template <class MatrixType>  TMatrix<MatrixType>::TMatrix(const TVector<TVector<MatrixType> > &mt):  TVector<TVector<MatrixType> >(mt) {}  template <class MatrixType>  bool TMatrix<MatrixType>::operator==(const TMatrix<MatrixType> &mt) const  {  return TVector<TVector<MatrixType> >::operator==(mt);  }  template <class MatrixType>  bool TMatrix<MatrixType>::operator!=(const TMatrix<MatrixType> &mt) const  {  return TVector<TVector<MatrixType> >::operator!=(mt);  }  template <class MatrixType>  TMatrix<MatrixType>& TMatrix<MatrixType>::operator=(const TMatrix<MatrixType> &mt)  {  TVector<TVector<MatrixType> >::operator=(mt);  return \*this;  }  template <class MatrixType>  TMatrix<MatrixType> TMatrix<MatrixType>::operator+(const TMatrix<MatrixType> &mt)  {  return TVector<TVector<MatrixType> >::operator+(mt);  }  template <class MatrixType>  TMatrix<MatrixType> TMatrix<MatrixType>::operator-(const TMatrix<MatrixType> &mt)  {  return TVector<TVector<MatrixType> >::operator-(mt);  }  template <class MatrixType>  TMatrix<MatrixType> TMatrix<MatrixType>::operator\*(const TMatrix<MatrixType> &mt)  {  if ( this -> size != mt.size )  throw 5;  TMatrix<MatrixType> buf ( this -> size );  for ( int row = 0; row < this -> size; row++ )  for ( int col = row; col < this -> size; col++ )  {  buf.pVector[row][col] = 0;  for ( int inner = row; inner <= col; inner++ )  buf.pVector[row][col] += this->pVector[row][inner] \* mt.pVector[inner][col];  }  return buf;  }  template <class MatrixType>  TMatrix<MatrixType> TMatrix<MatrixType>::inverse\_matrix() const  {  TMatrix<MatrixType> buf( this -> size );  for ( int i = 0; i < this->size; i++ )  for ( int j = i; j <this->size; j++ )  {  if ( i == j )  buf.pVector[i][j]=1;  else  buf.pVector[i][j]=0;  }  for ( int i = this -> size - 1; i >= 0; i-- )  {  for( int j = i-1; j >= 0; j-- )  {  for ( int inv = i; inv < this -> size; inv++ )  buf.pVector[j][inv] -= buf.pVector[i][inv] \* this->pVector[j][i] / this->pVector[i][i];  }  }  for ( int i = 0; i < this -> size; i++ )  for( int j = i; j < this -> size; j++ )  {  buf.pVector[i][j] /= this -> pVector[i][i];  }  return buf;  }  template <class MatrixType>  TMatrix<MatrixType> TMatrix<MatrixType>::operator/(const TMatrix<MatrixType> &mt)  {  if ( this -> size != mt.size )  throw 5;  TMatrix<MatrixType> buf (\*this);  TMatrix<MatrixType> buf2 ();  buf = buf \* mt.inverse\_matrix();  return buf;  } |

## 8.3. Приложение 3:Код программы тестирования и экспериментов

|  |
| --- |
| **sample\_matrix.cpp** |
| #include <iostream>  #include "utmatrix.cpp"  #include "utvector.cpp"  int main()  {  TMatrix<int> a(5), b(5), c(5);  int i, j;  std::cout<<"Matrix addition example"<<std::endl;  for ( i = 0; i < 5; i++ )  for ( j = i; j < 5; j++ )  {  a[i][j] = i \* 10 + j;  b[i][j] = ( i \* 10 + j ) \* 100;  }  c = a + b;  cout << "Matrix a = " << endl << a << endl;  cout << "Matrix b = " << endl << b << endl;  cout << "Matrix c = a + b" << endl << c << endl;  return 0;  } |

|  |
| --- |
| **sample\_input\_output.cpp** |
| #include <iostream>  #include "utmatrix.cpp"  #include "utvector.cpp"  int main()  {  TMatrix<int> a(5);  std::cout<<"Enter matrix with size = 5:"<<std::endl;  cin>>a;  /\* Enter something like:  \* 1 2 3 4 5 press Enter  \* 4 3 2 1 press Enter  \* 1 2 3 press Enter  \* 2 1 press Enter  \* 1 press Enter  \*/  cout<<"Matrix: "<<std::endl;  cout<<a;  /\* You will see:  \* 1 2 3 4 5  \* 0 4 3 2 1  \* 0 0 1 2 3  \* 0 0 0 2 1  \* 0 0 0 0 1  \*/  return 0;  } |

|  |
| --- |
| **sample\_performance\_check.cpp** |
| #include <iostream>  #include "utmatrix.cpp"  #include "utvector.cpp"  #include "time.h"  using namespace std;  int main()  {  int max\_count = 0;  cout << "Enter number of checks - ";  cin >> max\_count;  clock\_t time;  clock\_t average\_time;  for ( unsigned size = 10; size < 100000; size \*= 10 )  {  TMatrix<int> matrix1(size);  TMatrix<int> matrix2(size);  for ( int i = 0; i < size; i++ )  for ( int j = i; j < size; j++ )  {  matrix1[i][j] = i;  matrix2[i][j] = j;  }  average\_time = 0;  for (unsigned count = 0; count < max\_count; count++)  {  time = clock();  matrix1 + matrix2;  average\_time += clock() - time;  }  average\_time /= max\_count;  cout<< "Time is: " <<average\_time<<endl;  }  return 0;  } |

## 8.4. Приложение 4:Тесты для классов

|  |
| --- |
| **test\_main.cpp** |
| #include <gtest.h>  int main(int argc, char \*\*argv)  {  ::testing::InitGoogleTest(&argc, argv);  return RUN\_ALL\_TESTS();  } |

|  |
| --- |
| **test\_tvector.cpp** |
| #include "utmatrix.cpp"  #include "utvector.cpp"  #include <gtest.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, 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> v(10);  for (int i = 0; i<10;i++){  v[i]=i;  }  TVector<int> v2=v;  EXPECT\_EQ(v2,v);  }  TEST(TVector, copied\_vector\_has\_its\_own\_memory)  {  TVector<int> v(10);  TVector<int> v2=v;  v2[4]=333;  EXPECT\_NE(v2,v);  }  TEST(TVector, can\_get\_size)  {  TVector<int> v(4);  EXPECT\_EQ(4, v.GetSize());  }  TEST(TVector, can\_get\_start\_index)  {  TVector<int> v(4, 2);  EXPECT\_EQ(2, v.GetStartIndex());  }  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> v(4);  ASSERT\_ANY\_THROW(v[-2]=0);  }  TEST(TVector, throws\_when\_set\_element\_with\_too\_large\_index)  {  TVector<int> v(4);  ASSERT\_ANY\_THROW(v[7]=0);  }  TEST(TVector, can\_assign\_vector\_to\_itself)  {  TVector<int> v(2);  v[0]=1;  v[1]=2;  v=v;  EXPECT\_EQ(v[0],1);  EXPECT\_EQ(v[1],2);  }  TEST(TVector, can\_assign\_vectors\_of\_equal\_size)  {  TVector<int> v1(2),v2(2);  v1[0]=1;  v1[1]=2;  v2=v1;  EXPECT\_EQ(v2[0],1);  EXPECT\_EQ(v2[1],2);  }  TEST(TVector, assign\_operator\_change\_vector\_size)  {  TVector<int> v1(3),v2(2);  v1[0]=1;  v1[1]=2;  v2=v1;  EXPECT\_EQ(v2.GetSize(),3);  }  TEST(TVector, can\_assign\_vectors\_of\_different\_size)  {  TVector<int> v1(2),v2(5);  v1[0]=1;  v1[1]=2;  v2=v1;  EXPECT\_EQ(v2[0],1);  EXPECT\_EQ(v2[1],2);  }  TEST(TVector, compare\_equal\_vectors\_return\_true)  {  TVector<int> v1(2),v2(2);  v1[0]=1;  v1[1]=2;  v2=v1;  EXPECT\_TRUE(v2==v1);  }  TEST(TVector, compare\_vector\_with\_itself\_return\_true)  {  TVector<int> v1(2);  EXPECT\_TRUE(v1==v1);  }  TEST(TVector, vectors\_with\_different\_size\_are\_not\_equal)  {  TVector<int> v1(2),v2(3);  EXPECT\_FALSE(v2==v1);  }  TEST(TVector, can\_add\_scalar\_to\_vector)  {  TVector<int> v1(2),v2(2);  v1[0]=0;v1[1]=0;  v2=v1+7;  EXPECT\_EQ(v2[0],7);  EXPECT\_EQ(v2[1],7);  }  TEST(TVector, can\_subtract\_scalar\_from\_vector)  {  TVector<int> v1(2),v2(2);  v1[0]=0;v1[1]=0;  v2=v1-7;  EXPECT\_EQ(v2[0],-7);  EXPECT\_EQ(v2[1],-7);  }  TEST(TVector, can\_multiply\_scalar\_by\_vector)  {  TVector<int> v1(2),v2(2);  v1[0]=1;v1[1]=1;  v2=v1\*7;  EXPECT\_EQ(v2[0],7);  EXPECT\_EQ(v2[1],7);  }  TEST(TVector, can\_add\_vectors\_with\_equal\_size)  {  TVector<int> v1(3),v2(3),v(3);  v1[0]=3; v1[1]=2; v1[2]=1;  v2[0]=3; v2[1]=2; v2[2]=3;  v=v1+v2;  EXPECT\_EQ(v[0],6);  EXPECT\_EQ(v[1],4);  EXPECT\_EQ(v[2],4);  }  TEST(TVector, cant\_add\_vectors\_with\_not\_equal\_size)  {  TVector<int> v1(3),v2(4),v(3);  v1[0]=3; v1[1]=2; v1[2]=1;  v2[0]=3; v2[1]=2; v2[2]=3; v2[3]=1;  ASSERT\_ANY\_THROW( v=v1+v2);  }  TEST(TVector, can\_subtract\_vectors\_with\_equal\_size)  {  TVector<int> v1(3),v2(3),v(3);  v1[0]=3; v1[1]=2; v1[2]=1;  v2[0]=3; v2[1]=1; v2[2]=3;  v=v1-v2;  EXPECT\_EQ(v[0],0);  EXPECT\_EQ(v[1],1);  EXPECT\_EQ(v[2],-2);  }  TEST(TVector, cant\_subtract\_vectors\_with\_not\_equal\_size)  {  TVector<int> v1(3),v2(4),v(3);  v1[0]=3; v1[1]=2; v1[2]=1;  v2[0]=3; v2[1]=2; v2[2]=3; v2[3]=1;  ASSERT\_ANY\_THROW( v=v1-v2);  }  TEST(TVector, can\_multiply\_vectors\_with\_equal\_size)  {  TVector<int> v1(3),v2(3);  int v;  v1[0]=3; v1[1]=2; v1[2]=2;  v2[0]=3; v2[1]=1; v2[2]=3;  v=v1\*v2;  EXPECT\_EQ(v,17);  }  TEST(TVector, cant\_multiply\_vectors\_with\_not\_equal\_size)  {  TVector<int> v1(3),v2(4),v(3);  v1[0]=3; v1[1]=2; v1[2]=1;  v2[0]=3; v2[1]=2; v2[2]=3; v2[3]=1;  ASSERT\_ANY\_THROW( v=v1\*v2);  } |

|  |
| --- |
| **test\_tmatrix.cpp** |
| #include "utmatrix.cpp"  #include "utvector.cpp"  #include <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, throws\_when\_create\_matrix\_with\_negative\_length)  {  ASSERT\_ANY\_THROW(TMatrix<int> m(-5));  }  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> c = m;  EXPECT\_EQ(m, c);  }  TEST(TMatrix, copied\_matrix\_has\_its\_own\_memory)  {  TMatrix<int> m(5);  TMatrix<int> c = m;  c[1][1]=5;  EXPECT\_NE(m,c);  }  TEST(TMatrix, can\_get\_size)  {  TMatrix<int> m(4);  EXPECT\_EQ(m.GetSize(), 4);  }  TEST(TMatrix, can\_set\_and\_get\_element)  {  TMatrix<int> m(5);  m[1][1]=5;  EXPECT\_EQ(m[1][1],5);  }  TEST(TMatrix, throws\_when\_set\_element\_with\_negative\_index)  {  TMatrix<int> m(5);  ASSERT\_ANY\_THROW(m[-1][1]=5;);  }  TEST(TMatrix, throws\_when\_set\_element\_with\_too\_large\_index)  {  TMatrix<int> m(5);  ASSERT\_ANY\_THROW(m[5][5]=5;);  }  TEST(TMatrix, can\_assign\_matrix\_to\_itself)  {  TMatrix<int> m(5);  m[2][2]=1;  m[1][1]=2;  m=m;  EXPECT\_EQ(m[2][2],1);  EXPECT\_EQ(m[1][1],2);  }  TEST(TMatrix, can\_assign\_matrices\_of\_equal\_size)  {  TMatrix<int> m(5),m2(5);  m[2][2]=1;  m[1][1]=2;  m2=m;  EXPECT\_EQ(m2[2][2],1);  EXPECT\_EQ(m2[1][1],2);  }  TEST(TMatrix, assign\_operator\_change\_matrix\_size)  {  TMatrix<int> m(5),m2(10);  m2=m;  EXPECT\_EQ(m2.GetSize(),5);  }  TEST(TMatrix, can\_assign\_matrices\_of\_different\_size)  {  TMatrix<int> m(5),m2(7);  m[2][2]=1;  m[1][1]=2;  m2=m;  EXPECT\_EQ(m2[2][2],1);  EXPECT\_EQ(m2[1][1],2);  }  TEST(TMatrix, compare\_equal\_matrices\_return\_true)  {  TMatrix<int> m(5),m2(7);  m[2][2]=1;  m[1][1]=2;  m2=m;  EXPECT\_TRUE(m2==m);  }  TEST(TMatrix, compare\_matrix\_with\_itself\_return\_true)  {  TMatrix<int> m(5);  EXPECT\_TRUE(m==m);  }  TEST(TMatrix, matrices\_with\_different\_size\_are\_not\_equal)  {  TMatrix<int> m1(2), m2(3);  EXPECT\_FALSE(m1 == m2);  }  TEST(TMatrix, can\_add\_matrices\_with\_equal\_size)  {  TMatrix<int> m1(2), m2(2);  m1[0][0]=3; m1[1][1]=3;m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5;m2[0][1]=2;  TMatrix<int> m3(3);  m3=m2+m1;  EXPECT\_EQ(m3[0][0],8);  EXPECT\_EQ(m3[1][1],8);  EXPECT\_EQ(m3[0][1],4);  }  TEST(TMatrix, cant\_add\_matrices\_with\_not\_equal\_size)  {  TMatrix<int> m1(3), m2(2);  m1[0][0]=3; m1[1][1]=3;m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5;m2[0][1]=2;  ASSERT\_ANY\_THROW( m2+m1);  }  TEST(TMatrix, can\_subtract\_matrices\_with\_equal\_size)  {  TMatrix<int> m1(2), m2(2);  m1[0][0]=3; m1[1][1]=3;m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5;m2[0][1]=2;  TMatrix<int> m3(3);  m3=m2-m1;  EXPECT\_EQ(m3[0][0],2);  EXPECT\_EQ(m3[1][1],2);  EXPECT\_EQ(m3[0][1],0);  }  TEST(TMatrix, cant\_subtract\_matrixes\_with\_not\_equal\_size)  {  TMatrix<int> m1(3), m2(2);  m1[0][0]=3; m1[1][1]=3;m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5;m2[0][1]=2;  ASSERT\_ANY\_THROW( m2-m1);  }  TEST(TMatrix, cant\_multiply\_matrixes\_with\_not\_equal\_size)  {  TMatrix<int> m1(3), m2(2);  m1[0][0]=3; m1[1][1]=3; m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5; m2[0][1]=2;  ASSERT\_ANY\_THROW( m2\*m1);  }  TEST(TMatrix, can\_multiply\_matrixes\_with\_equal\_size)  {  TMatrix<int> m1(2), m2(2), m3(2);  m1[0][0]=1; m1[1][1]=2; m1[0][1]=3;  m2[0][0]=1; m2[1][1]=2; m2[0][1]=3;  m3[0][0]=1; m3[1][1]=4; m3[0][1]=9;  EXPECT\_EQ(m3, (m2 \* m1));  }  TEST(TMatrix, cant\_divide\_matrixes\_with\_not\_equal\_size)  {  TMatrix<double> m1(3), m2(2);  m1[0][0]=3; m1[1][1]=3; m1[0][1]=2;  m2[1][1]=5; m2[0][0]=5; m2[0][1]=2;  ASSERT\_ANY\_THROW( m2 / m1 );  }  TEST(TMatrix, can\_divide\_matrixes\_with\_equal\_size)  {  TMatrix<double> m1(2), m2(2), m3(2), m4(2);  m1[0][0]=2; m1[0][1]=2; m1[1][1]=2;  m2[0][0]=2; m2[0][1]=2; m2[1][1]=2;  m3 = m1 / m2;  m4[0][0]=1; m4[0][1]=0; m4[1][1]=1;  EXPECT\_EQ(m3, m4);  } |