Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

"Национальный исследовательский Нижегородский государственный

университет им. Н. И. Лобачевского"

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

СТРУКТУРА ХРАНЕНИЯ МАТРИЦ

Выполнил: студент группы 381808-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шульман Е. А.

Подпись

Проверил:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А. А.

Подпись

Нижний Новгород

2019

Содержание

[**Введение** 3](#_Toc23539888)

[**Постановка задачи** 5](#_Toc23539889)

[**Руководство пользователя** 6](#_Toc23539890)

[**Руководство программиста** 9](#_Toc23539891)

[Описание структуры программы 9](#_Toc23539892)

[Описание структуры данных 10](#_Toc23539893)

[Описание алгоритмов 11](#_Toc23539894)

[**Заключение** 14](#_Toc23539895)

[**Литература** 15](#_Toc23539896)

[**Приложения** 16](#_Toc23539897)

[Приложение 1 16](#_Toc23539898)

[Приложение 2 17](#_Toc23539899)

[Приложение 3 18](#_Toc23539900)

[Фрагменты исходного кода программы 19](#_Toc23539901)

[Класс TVector 19](#_Toc23539902)

[Класс TMatrix 22](#_Toc23539903)

**Введение**

Термин «Матрица» имеет много значений.

Например, в математике матрицей называется система элементов, имеющая вид прямоугольной таблицы, в программировании матрица – это двумерный массив, в электронике – набор проводников, которые можно замкнуть в точках их пересечений. Покерные фишки также имеют непосредственное отношение к матрице. Фишки для покера изготавливаются из высококачественного композиционного материала, зачастую с металлической сердцевиной. В свою очередь композиционный материал или композит имеет матрицу и включенные в нее армирующие элементы (исключение составляют слоистые композиты). Матрица в фотографии – это интегральная микросхема (аналоговая или цифро-аналоговая), которая состоит из фотодиодов (светочувствительных элементов). Благодаря светочувствительной матрице происходит преобразование спроецированного на нее оптического изображения в электрический сигнал аналогового типа, а при наличии в составе матрицы АЦП, то преобразование происходит в поток цифровых данных. Матрица – основной элемент цифровых фотоаппаратов, всех современных видео- и телекамер, фотокамер, встроенных в мобильный телефон и системы видеонаблюдения.

Основное значение термин «матрица» имеет в математике.

Матрица — математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов матрицы задают размер матрицы. Хотя исторически рассматривались, например, треугольные матрицы, в настоящее время говорят исключительно о матрицах прямоугольной формы, так как они являются наиболее удобными и общими.

Впервые матрицы упоминались ещё в древнем Китае, называясь тогда «волшебным квадратом». Основным применением матриц было решение линейных уравнений. Так же, волшебные квадраты были известны чуть позднее у арабских математиков, примерно тогда появился принцип сложения матриц. После развития теории определителей в конце 17-го века, Габриэль Крамер начал разрабатывать свою теорию в 18-ом столетии и опубликовал «правило Крамера» в 1751 году. Примерно в этом же промежутке времени появился «метод Гаусса». Теория матриц начала своё существование в середине XIX века в работах Уильяма Гамильтона и Артура Кэли. Фундаментальные результаты в теории матриц принадлежат Вейерштрассу, Жордану, Фробениусу. Термин «матрица» ввел Джеймс Сильвестр в 1850 г.

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

**Постановка задачи**

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

* сложение/вычитание;
* умножение;
* копирование;
* сравнение.

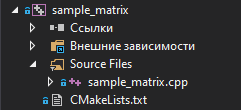
Программные средства должны содержать:

* класс Вектор (на шаблонах);
* класс Матрица (на шаблонах);

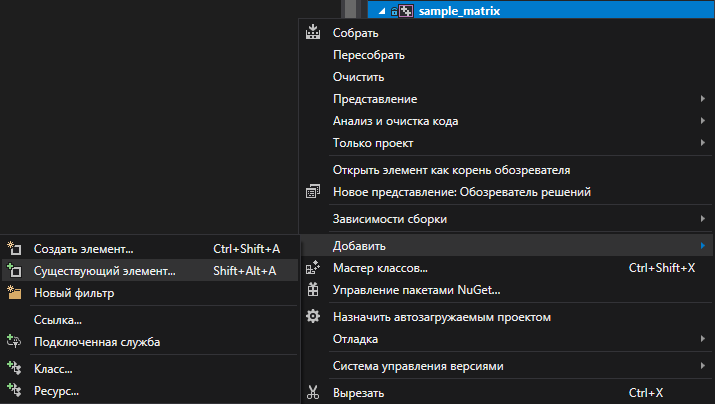
**Руководство пользователя**

По результатам работы был реализован заголовочный файл utmatrix.h для того, чтобы его использовать необходимо:

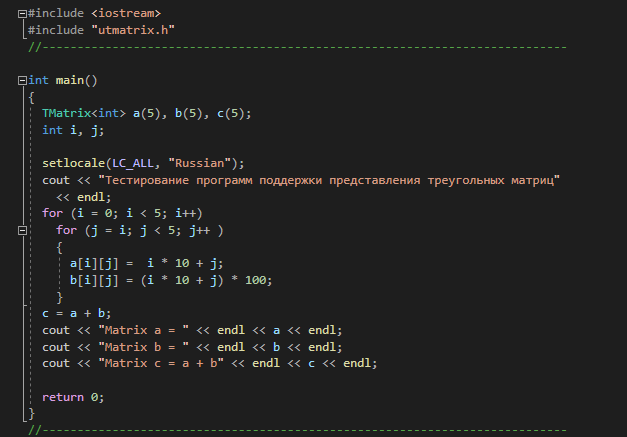
1. Создать новый проект.



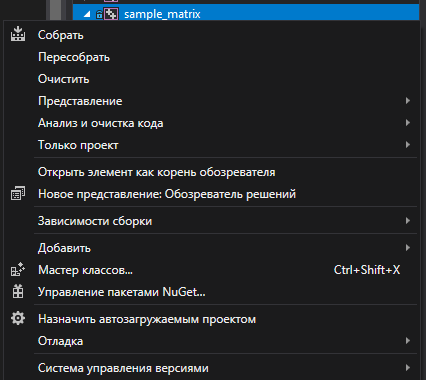
1. Добавляем существующий элемент и выбираем utmatrix.h.



1. Подключаем заголовочный файл.

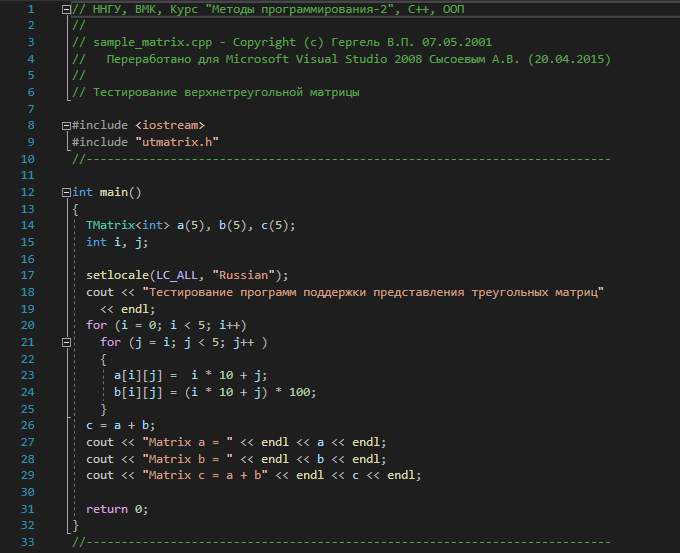


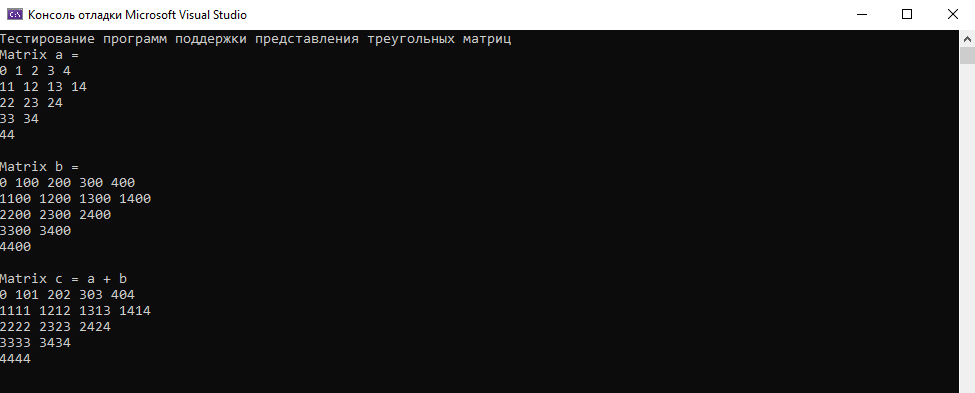
1. Назначаем наш проект автозагружаемым.



1. Теперь нашему проекту доступны функции utmatrix.h.

Проверим это:





**Руководство программиста**

Описание структуры программы

Программа состоит из 4 проектов:

* gtest – библиотека google тестов;
* sample\_matrix – тестовое приложение, позволяющее тестировать класс TMatrix;
* matrix – служебная программа, содержащая модуль utmatirx, содержащий реализацию классов TVector и TMatrix.
* test\_matrix – тесты для классов TVector и TMatrix;

Описание структуры данных

В данной лабораторной работе, мы храним матрицу в виде набора векторов разной длины:

Вектор

Матрица

Очевидно сходство в задании структуры данных Вектор, как набора элементов, связанных отношением следования, и структуры данных Матрица, как набора элементов векторов, связанных отношением следования. Этот факт позволяет единообразно организовать алгоритмы обработки векторов и матриц, а, следовательно, использовать при разработке требуемых классов механизм наследования.

То есть для реализации алгоритмов нами использовано 2 класса:

* Класс «Вектор» (TVector).
* Класс «Матрицы» (TMatrix), который будет использовать класс векторов.

Описание алгоритмов

Класс TVector

Наиболее сложные методы класса

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

Конструктор создает вектор длины s и стартовым индексом si;

Если [s < 0] или [s > MAX\_VECTOR\_SIZE] или [si < 0] вызывается исключение.

После выделения памяти всем элементам массива pVector присваивается 0.

* TVector (const TVector<ValType>& v);

Конструктор копирования создает новый вектор, копируя данные из существующего v.

* int GetSize();

Данный метод возвращает размер вектора.

* int GetStartIndex();

Данный метод возвращает индекс первого элемента.

* ValType& operator[](int pos);

Перегруженный оператор [] предоставляет доступ к элементу [pos – si].

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

Перегруженный оператор = копирует данные вектора v в \*this. Если [this = &v], то смысла выполнять присваивание нет - сразу возвращаем \*this. Если длины векторов различны – пересоздаем вектор this.

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

Перегруженный оператор == сравнивает левый и правый операнды и возвращает «1», если они равны, иначе возвращает «0». Если длины векторов или стартовые индексы различны – сразу возвращаем «0».

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

Перегруженный оператор != сравнивает левый и правый операнды и возвращает «0», если они равны, иначе возвращает «1». Если длины векторов или стартовые индексы различны – сразу возвращаем «1».

* TVector operator+(const ValType &val);

Прибавляет к каждому элементу вектора значение val.

* TVector operator-(const ValType &val);

Вычитает из каждого элемента вектора значение val.

* TVector operator\*(const ValType &val);

Умножает каждый элемент вектора на значение val.

* TVector operator+(const TVector &v);

Поэлементно складывает два вектора, если длины векторов

различны - вызывается исключение.

* TVector operator-(const TVector &v);

Из вектора this поэлементно вычитается вектор v, если длины векторов различны - вызывается исключение.

* TVector operator\*(const TVector &v);

Вычисляется скалярное произведение по формуле: ,если длины векторов различны - вызывается исключение.

* friend istream &operator>>(istream &in, TSet &v);

Перегруженный оператор >> позволяет вводить вектор.

* friend ostream &operator<<(ostream &out, const TSet & v);

Перегруженный оператор << позволяет выводить через пробел все элементы вектора.

Класс TMatrix

Наиболее сложные методы класса

* TMatrix (int s = 10);

Конструктор создает матрицу из s векторов.

При этом, с увеличением счетчика цикла создаются векторы меньшего размера – как и было задумано.

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

Перегруженный оператор == сравнивает левый и правый операнды и возвращает «1», если они равны, иначе возвращает «0». Если размер матриц различны – сразу возвращаем «0».

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

Перегруженный оператор != сравнивает левый и правый операнды и возвращает «0», если они равны, иначе возвращает «1». Если размер матриц различны – сразу возвращаем «1».

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

Перегруженный оператор = копирует данные матрицы mt в \*this. Если [this = & mt], то смысла выполнять присваивание нет - сразу возвращаем \*this. Если размеры матриц различны – пересоздаем матрицу this.

* TMatrix operator+ (const TMatrix &mt);

Построчно складывает две матрицы.

* TMatrix operator- (const TMatrix &mt);

Построчно вычитает из матрицы this матрицу mt.

* friend istream &operator>>(istream &in, TSet &mt);

Перегруженный оператор >> позволяет вводить матрицу.

* friend ostream &operator<<(ostream &out, const TSet &mt);

Перегруженный оператор << позволяет выводить матрицу.

**Заключение**

Целью работы была реализация структуры данных для хранения матриц специального вида (верхнетреугольных матриц).

В результате была написана структура данных для хранения матриц специального вида (верхнетреугольных матриц), в которой уменьшены затраты памяти для хранения векторов, путем создания векторов разной длины. Были переопределены арифметические операции для работы с матрицами, реализованными в данной структуре.

Программные средства созданы при помощи среды программирования С++ и содержат:

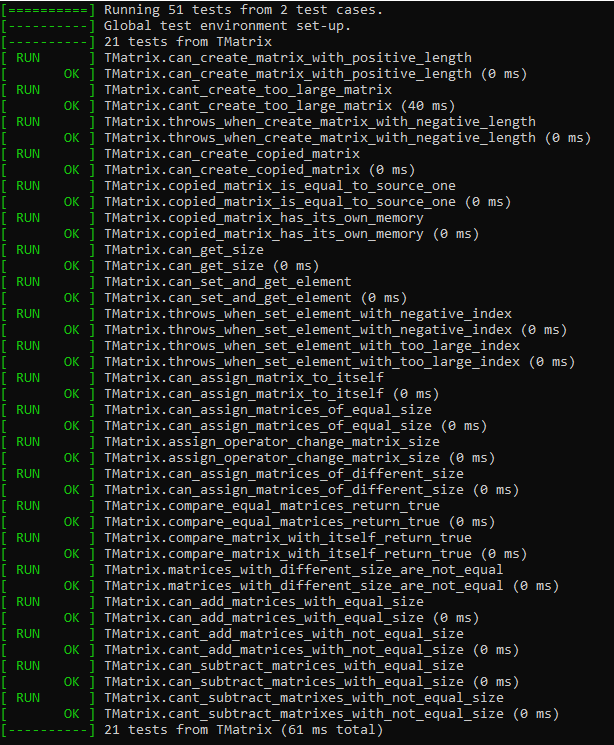
* класс Вектор (на шаблонах);
* класс Матрица (на шаблонах);

В программе также реализована система обработки ошибок.

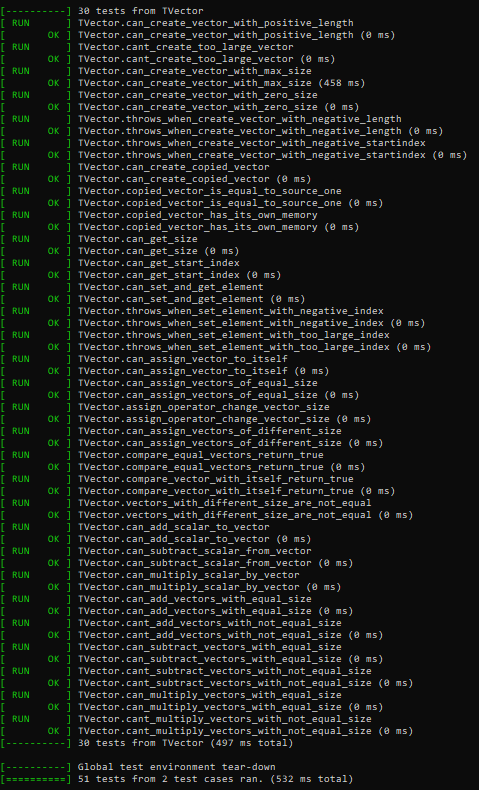
**Литература**

**Приложения**

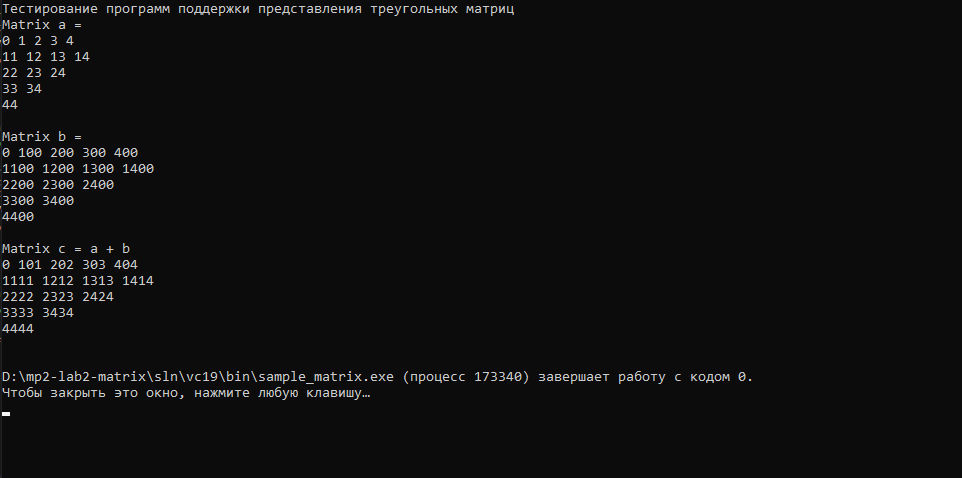
Приложение 1



Приложение 2



Приложение 3



## Фрагменты исходного кода программы

Класс TVector

// 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)

throw "Error";

if (si < 0)

throw "Error";

Size = s;

StartIndex = si;

pVector = new ValType[Size];

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

pVector[i] = (ValType)0;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> //конструктор копирования

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

{

Size = v.Size;

StartIndex = v.StartIndex;

pVector = new ValType[Size];

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

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

} /\*-------------------------------------------------------------------------\*/

template <class ValType>

TVector<ValType>::~TVector()

{

if (pVector != NULL)

{

delete[] pVector;

pVector = NULL;

}

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // доступ

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

{

if (pos < 0 || pos >= StartIndex + Size)

throw "Error";

return pVector[pos - StartIndex];

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // сравнение

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

{

if (Size != v.Size)

return false;

if(StartIndex != v.StartIndex)

return false;

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

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

return false;

return true;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // сравнение

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

{

if (Size != v.Size)

return true;

if (StartIndex != v.StartIndex)

return true;

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

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

return true;

return false;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // присваивание

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

{

if (this != &v)

{

if (Size != v.Size)

{

delete[] pVector;

pVector = new ValType[v.Size];

Size = v.Size;

}

StartIndex = v.StartIndex;

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

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

}

return \*this;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // прибавить скаляр

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

{

TVector<ValType> tmp(\*this);

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

tmp.pVector[i] = tmp.pVector[i] + val;

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // вычесть скаляр

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

{

TVector<ValType> tmp(\*this);

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

tmp.pVector[i] = tmp.pVector[i] - val;

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // умножить на скаляр

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

{

TVector<ValType> tmp(\*this);

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

tmp.pVector[i] = tmp.pVector[i] \* val;

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // сложение

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

{

if (Size != v.Size)

throw "Error";

TVector<ValType> tmp(\*this);

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

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

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // вычитание

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

{

if (Size != v.Size)

throw "Error";

TVector<ValType> tmp(\*this);

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

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

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // скалярное произведение

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

{

if (Size != v.Size)

throw "Error";

ValType tmp = 0;

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

tmp += pVector[i] \* v.pVector[i];

return tmp;

} /\*-------------------------------------------------------------------------\*/

Класс TMatrix

// ввод / вывод

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)

{

// по умолчанию создается квадратная матрица sхs

// надо заменить созданную матрицу верхнетреугольной

// вектора должны быть разной длины в матрице

if (s > MAX\_MATRIX\_SIZE)

throw "Error";

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

{

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

pVector[i] = t;

}

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // конструктор копирования

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

TVector<TVector<ValType> >(mt) {}

template <class ValType> // конструктор преобразования типа

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

TVector<TVector<ValType> >(mt) {}

template <class ValType> // сравнение

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

{

if(Size != mt.Size)

return false;

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

if(pVector[i] != mt.pVector[i])

return false;

return true;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // сравнение

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

{

if (Size != mt.Size)

return true;

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

if (pVector[i] != mt.pVector[i])

return true;

return false

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // присваивание

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

{

if (this != &mt)

{

if (Size != mt.Size)

{

Size = mt.Size;

delete[] pVector;

pVector = new TVector<ValType>[mt.Size];

}

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

pVector[i] = mt.pVector[i];

}

return \*this;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // сложение

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

{

TMatrix<ValType> tmp(\*this);

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

tmp.pVector[i] = pVector[i] + mt.pVector[i];

return tmp;

} /\*-------------------------------------------------------------------------\*/

template <class ValType> // вычитание

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

{

TMatrix<ValType> tmp(\*this);

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

tmp.pVector[i] = pVector[i] - mt.pVector[i];

return tmp;

} /\*--------------------------------------------------s-----------------------\*/