Министерство науки и высшего образования 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 №2**

**«Структуры хранения матриц специального вида»**

Выполнила:

студентка ПМИ – 381903-3

Страшенко Мария

Проверил:

ассистент кафедры МОСТ

Лебедев Илья Геннадьевич

Нижний Новгород

2020

Содержание

[Введение 3](#_Toc23876641)

[1. Постановка задачи 4](#_Toc23876642)

[2. Руководство пользователя 5](#_Toc23876643)

[3. Руководство программиста 6](#_Toc23876644)

[3.1. Описание структуры программы 6](#_Toc23876645)

[3.2. Описание алгоритмов 7](#_Toc23876646)

[Заключение 10](#_Toc23876647)

[Литература 11](#_Toc23876648)

[Приложение 12](#_Toc23876649)

# Введение

Понятие Матрица в европейской науке было введено в работах У. Гамильтона3 и А. Кэли4 в середине XIX века. 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.

# Постановка задачи

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

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

Программные средства должны содержать:

* класс Вектор (с использованием шаблона);
* класс Матрица (с использованием шаблона);

# Руководство пользователя

Задаётся матрицы A и B. Их значения выводятся на экран. Потом они складываются.

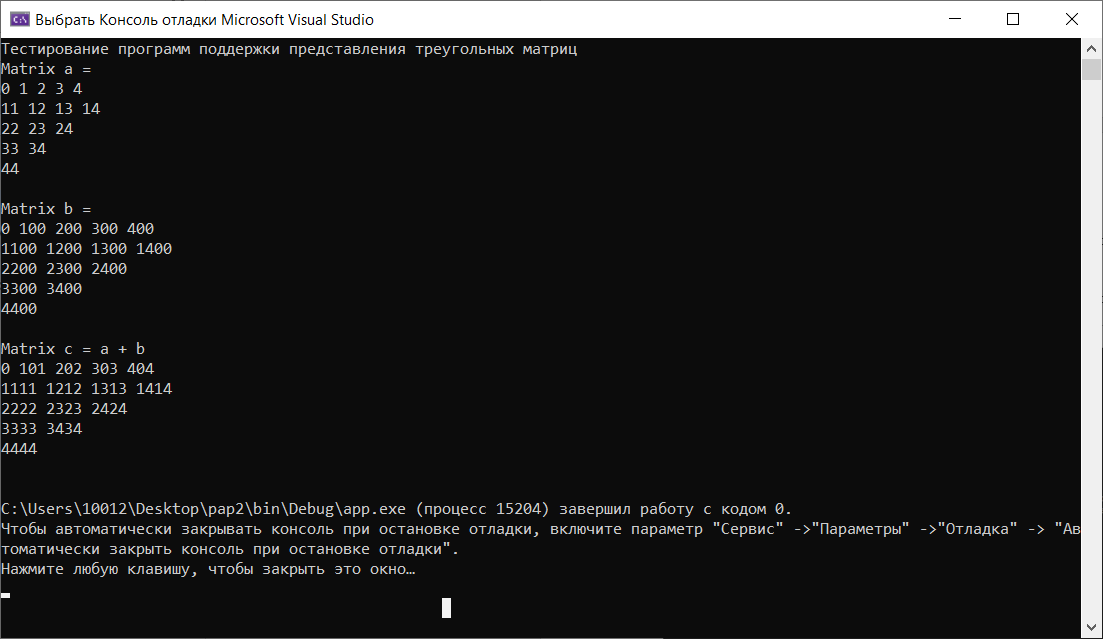


Рис. 1. Тестирование программы

## Руководство программиста

## Описание структуры программы

Программа написана на языке программирования C++.

Состоит из пяти файлов:

**TVector.h** – класс векторов и методы реализации класса векторов,,

**TMatrix.h** – класс матриц и методы реализации класса матриц,

**main.cpp** – файл тестирования.

В программе реализованы следующие классы:

1. **Класс «TVector»**

* Описание: структура данных Вектор.

|  |
| --- |
| **TVector** |
| ValType \*pVector;  int size;  int startIndex; |
| “**constructor**”  TVector(int s = 10, int si = 0);  TVector(const TVector &v);  “**interface**”  int GetSize() { return size; };  int GetStartIndex() { return startIndex; };  ValType& GetValue(int pos);  ValType& operator[](int pos);  bool operator==(const TVector &v);  TVector& operator= (const TVector &v);  TVector operator+ (const ValType &val);  TVector operator- (const ValType &val);  TVector operator\* (const ValType &val);  TVector operator+ (const TVector &v);  TVector operator- (const TVector &v);  ValType operator\* (const TVector &v);  friend istream& operator>>(istream &in, TVector &v);  friend ostream& operator<<(ostream &out, const TVector &v); |

В следующей таблице указано применение данных и методов класса.

|  |  |  |
| --- | --- | --- |
| **№** | **Данные** | **Хранит** |
| 1 | pVector | Память под вектор |
| 2 | size | Размер вектора |
| 3 | startIndex; | Индекс первого элемента вектора |
| **№** | **Методы** | **Применение** |
| 1 | GetSize | Получить размер вектора |
| 2 | GetStartIndex | Индекс первого элемента |
| 3 | GetValue | Доступ с контролем индекса |
| **№** | **Конструк./Деструк.** | **Применение** |
| 1 | TVector(int s = 10, int si = 0); | Конструктор инициализации |
| 2 | TVector(const TVector &v); | Конструктор копирования |
| 3 | ~TVector(); | Деструктор |
| **№** | **Операторы** | **Применение** |
| 1 | operator+ (const ValType &val); | Прибавить скаляр |
| 2 | operator- (const ValType &val); | Вычесть скаляр |
| 3 | operator\* (const ValType &val); | Умножить на скаляр |
| 4 | operator+ (const TVector &v); | Сложение векторов |
| 5 | operator- (const TVector &v); | Вычитание векторов |
| 6 | operator\* (const TVector &v); | Скалярное произведение |
| 7 | Operator [] | Оператор доступ |
| 8 | Operator >> | Оператор ввод |
| 9 | Operator << | Оператор вывод |

1. **Класс «TMatrix»**

Описание: структура данных Матрица.

|  |
| --- |
| **TMatrix** |
|  |
| “**constructor**”  TMatrix(int s = 10);  TMatrix(const TMatrix &mt);  TMatrix(const TVector<TVector<ValType> > &mt);  “**interface**”  bool operator==(const TMatrix &mt);  TMatrix& operator= (const TMatrix &mt);  TMatrix operator+ (const TMatrix &mt);  TMatrix operator- (const TMatrix &mt);  TMatrix operator\* (const TMatrix &mt);  friend istream& operator>>(istream &in, TMatrix &mt)  friend ostream & operator<<(ostream &out, const TMatrix &mt) |

В следующей таблице указано применение данных и методов класса.

|  |  |  |
| --- | --- | --- |
| **№** | **Конструк./Деструк.** | **Применение** |
| 1 | TMatrix(int s = 10) | Конструктор инициализации |
| 2 | TMatrix(const TMatrix &mt); | Конструктор копирования |
| 3 | TMatrix(const TVector<TVector<ValType> > &mt); | Конструктор преобразования типа |
| **№** | **Операторы** | **Применение** |
| 1 | Operator = | Оператор присваивания |
| 2 | Operator = = | Оператор сравнения |
| 3 | Operator \* | Оператор умножения |
| 4 | Operator >> | Оператор ввод |
| 5 | Operator << | Оператор вывод |

## Описание алгоритмов

В классе «TVector» реализованы методы для работы с векторами.

1. Метод «GetValue»

Получение значения указанного элемента в векторе. Это происходит путём нахождения разницы указанной позиции и начального индекса в векторе.

**Тип**& TVector<**Тип**>:: **Функция GetValue**(**цел** pos) {

**если** (pos < 0)

throw **логическое исключение**("Input error: index cannot be negative value.");

**если** (startIndex > pos)

throw **исключение выхода из диапазона**("Input error: index is out of range, less then start index.");

**если** (pos >= size + startIndex)

throw **исключение выхода из диапазона** ("Input error: index is out of range, more then size.");

**вернуть** pVector[pos - startIndex];

}

1. Оператор “Operator+”

Сложение происходит путём прибавления аргумента к значению всех элементов вектора. Возвращает вектор.

TVector<**символ**> TVector<**символ**>::**Оператор operator+(**const ValType &val) {

TVector<**символ**> tmp(size, startIndex);

**цел** i;

**цикл** (i = 0; i < size; ++i)

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

**вернуть** tmp;

}

В классе «TMatrix» реализованы методы для работы с матрицами.

1. Оператор “Operator=”

Происходит присваивание вектора. Если размеры и стартовые индексы двух векторов не соответствуют происходит их присваивания этих переменных. Если равны, то сразу происходит присваивание значений одного вектора другому.

TMatrix<**символ**>& TMatrix<**символ**>::**Оператор operator=** (const TMatrix<**символ**> &mt) {

**если** (**не**(\*this == mt)) {

**цел** i;

**если** (**не**(this->size == mt.size)) {

(\*this).size = mt.size;

(\*this).startIndex = mt.startIndex;

}

**цикл** (i = 0; i < this->size; ++i)

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

} **вернуть** \*this;

# Заключение

При выполнении лабораторной работы реализовано хранение и операции с векторами и матрицами TVector и TMatrix.

# Литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. – СПБ.: Питер, 2018. – 928 с.
2. Страуструп Бьерн Язык программирования C++. Специальное издание. Пер. с англ. – М.: Издательство Бином, 2017 г. – 1136 с.
3. Шилдт Г. С++ Базовый курс. 3-е изд. – М.: Издательство Вильямс, 2018. 624 с.

# Приложение

TVector.h

#ifndef \_TVECTOR\_H\_

#define \_TVECTOR\_H\_

#include <iostream>

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();

int GetSize() { return Size; }

int GetStartIndex() { return StartIndex; }

ValType & GetValue(int pos);

ValType& operator[](int pos);

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

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

TVector& operator=(const TVector& v);

TVector operator+(const ValType& val);

TVector operator-(const ValType& val);

TVector operator\*(const ValType& val);

TVector operator+(const TVector& v);

TVector operator-(const TVector& v);

ValType operator\*(const TVector& v);

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

{

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

in >> v.pVector[i];

return in;

}

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

{

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

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

return out;

}

};

template<typename ValType>

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

: Size(s), StartIndex(si), pVector(nullptr)

{

if (Size <= 0)

throw "Invalid vector size";

pVector = new ValType[Size];

}

template<typename ValType>

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

: Size(v.Size), StartIndex(v.StartIndex), pVector(nullptr)

{

pVector = new ValType[Size];

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

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

}

template<typename ValType>

TVector<ValType>::~TVector()

{

if (pVector != nullptr)

delete[] pVector;

}

template<typename ValType>

bool TVector<ValType>::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<typename ValType>

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

{

return !(\*this == v);

}

template<typename ValType>

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

{

if (this != &v)

{

if (Size != v.Size)

{

Size = v.Size;

delete[] pVector;

pVector = new ValType[Size];

}

StartIndex = v.StartIndex;

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

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

}

return \*this;

}

template<typename ValType>

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

{

TVector<ValType> result(\*this);

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

result.pVector[i] += val;

return result;

}

template<typename ValType>

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

{

TVector<ValType> result(\*this);

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

result.pVector[i] -= val;

return result;

}

template<typename ValType>

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

{

TVector<ValType> result(\*this);

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

result.pVector[i] \*= val;

return result;

}

template<typename ValType>

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

{

if (Size != v.Size)

throw "Different sizes of vectors";

TVector<ValType> result(\*this);

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

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

return result;

}

template<typename ValType>

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

{

if (Size != v.Size)

throw "Different sizes of vectors";

TVector<ValType> result(\*this);

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

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

return result;

}

template<typename ValType>

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

{

if (Size != v.Size)

throw "Different sizes of vectors";

ValType result(0);

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

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

return result;

}

template<typename ValType>

ValType& TVector<ValType>::GetValue(int pos)

{

if (pos == 1) throw 1;

if (pos - StartIndex >= Size)

throw "Invalid pos";

return pVector[pos - StartIndex];

}

template<typename ValType>

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

{

if (pos - StartIndex >= Size)

throw "Invalid pos";

return pVector[pos - StartIndex];

}

#endif

TMatrix.h

#ifndef \_TMATRIX\_H\_

#define \_TMATRIX\_H\_

#include "TVector.h"

template <class ValType>

class TMatrix : public TVector<TVector<ValType>>

{

using TVector<TVector<ValType>>::pVector;

using TVector<TVector<ValType>>::Size;

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);

TMatrix operator\* (const TMatrix &mt);

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

{

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

in >> mt.pVector[i];

return in;

}

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

{

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

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

return out;

}

};

template<typename ValType>

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

{

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

pVector[i] = TVector<ValType>(Size - i, i);

}

template<typename ValType>

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

: TVector<TVector<ValType>>(mt)

{

}

template<typename ValType>

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

: TVector<TVector<ValType>>(mt)

{

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

if (mt[i].GetSize() != Size - i)

throw "Matrix is non-convertible";

}

template<typename ValType>

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

{

if (Size != mt.Size)

return false;

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

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

return false;

return true;

}

template<typename ValType>

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

{

return !(\*this == mt);

}

template<typename ValType>

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 < mt.Size; i++)

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

}

return \*this;

}

template<typename ValType>

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

{

if (Size != mt.Size)

throw "Matrices have different sizes";

TMatrix<ValType> result(\*this);

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

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

return result;

}

template<typename ValType>

TMatrix<ValType> TMatrix<ValType>::operator\*(const TMatrix& mt)

{

int f = 0;

TMatrix result(\*this);

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

{

for (int j = i; j < result.Size; j++)

{

result[i][j] = 0;

f = j + 1;

for (int t = i; t < f; t++)

{

result[i][j] = result[i][j] + mt.pVector[t][j] \* (\*this)[i][t];

}

}

}

return result;

}

template<typename ValType>

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

{

if (Size != mt.Size)

throw "Matrices have different sizes";

TMatrix<ValType> result(\*this);

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

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

return result;

}

#endif

Main.cpp

#include <iostream>

#include "TMatrix.h"

int main()

{

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

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

for (int j = i; j < 5; j++)

{

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

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

}

c = a + b;

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

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

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

}