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

Отчёт по лабораторной работе

Структуры хранения матриц специального вида

Выполнил:

студент института ИТММ гр.381908-1

Труханов А. В.

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г

Нижний Новгород

2021 г.

Оглавление

[Введение 3](#_Toc72880405)

[Постановка задачи 4](#_Toc72880406)

[Руководство пользователя 5](#_Toc72880407)

[Руководство программиста 6](#_Toc72880408)

[Описание структуры программы 6](#_Toc72880409)

[Описание алгоритмов 7](#_Toc72880410)

[Эксперименты 8](#_Toc72880411)

[Заключение 9](#_Toc72880412)

[Литература 10](#_Toc72880413)

[Приложения 11](#_Toc72880414)

# Введение

Понятие Матрица в европейской науке было введено в работах У. Гамильтона и А. Кэли 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.

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 (верхнетреугольных) и выполнение основных операций над ними:

· сложение/вычитание;

· умножение;

· копирование;

· сравнение.

Программные средства должны содержать:

· класс Вектор (на шаблонах);

· класс Матрица (на шаблонах);

· тестовое приложение, позволяющее задавать матрицы и осуществлять основные операции над ними.

# Руководство пользователя

Пользователю нужно запустить файл main.exe.

Откроется консольное приложение для тестирования матриц.

Программа заполнит две матрица числами и выведет их в консоль, так же выведет результат сложения матриц.

Для повторного выполнения потребуется перезапустить программу.

# Руководство программиста

## Описание структуры программы

Класс TMatrix реализует операции:

TMatrix<T>& operator = (const TMatrix<T>& A) – оператор присваивания

TMatrix<T> operator + (const TMatrix<T>& A) – оператор сложения

TMatrix<T> operator - (const TMatrix<T>& A) – оператор вычитания

TMatrix<T> operator\*(TMatrix<T>& A) – оператор умножения

bool operator == (const TMatrix<T>& A) const – оператор сравнения

bool operator != (const TMatrix<T>& A) const – оператор сравнения

Класс TVector реализует следующие операции:

Vector<T> operator +(const T& \_v) – прибавить скаляр

Vector<T> operator -(const T& \_v) – вычесть скаляр

Vector<T> operator \*(const T& \_v) – умножить на скаляр

Vector<T>& operator =(const Vector<T>& \_v) – оператор присваивания

T& operator[] (const int index) – оператор индексации

bool operator ==(const Vector<T>& \_v) const – оператор сравнения

bool operator !=(const Vector<T>& \_v) const – оператор сравнения

Vector operator +(const Vector& \_v) – сложение векторов

Vector operator -(const Vector& \_v) – вычитание векторов

Vector operator \*(const Vector& \_v) – скалярное произведение

С учетом структуры данных целесообразной представляется следующая модульная структура программы:

• Vector.h, Vector.cpp – модуль, реализующий структуру данных Вектор;

• Matrix.h, Matrix.cpp – модуль, реализующий структуру данных Матрица;

•test\_matrix.cpp – модуль программы тестирования матриц.

• test\_vector.cpp – модуль программы тестирования векторов.

# Описание алгоритмов

Создание вектора:

* Инициализируем размер вектора
* Заполняем вектор

Прибавление скаляра к вектору:

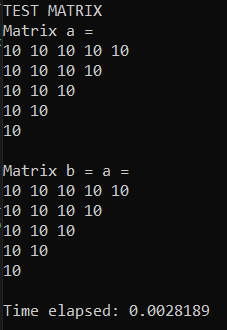
* Аргумент прибавляется к значению всех элементов вектора
* Возвращает вектор

Создание матрицы:

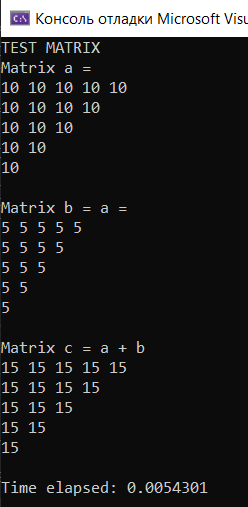
* Инициализируем строки матрицы
* Заполняем строки элементами от 0 до количества векторов, которые понадобятся для создания матрицы

# Эксперименты

Результат выполнения операции присваивания:



Результат выполнения операции сложения:



# Заключение

В ходе выполнения лабораторной работы были реализованы классы для работы с векторами и матрицами.

# Литература

1. Лабораторный практикум. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с.

# Приложения

MyVector.h

#ifndef \_MY\_VECTOR\_

#define \_MY\_VECTOR\_

#include <iostream>

using namespace std;

const int MAX\_VECTOR\_SIZE = 10000;

template <class T>

class Vector

{

protected:

int length;

T\* x;

int ind;

public:

Vector<T>\* vec;

Vector(int \_v=0, int \_ind=0);

Vector(const Vector<T>& \_v);

virtual ~Vector();

Vector<T> operator +(const T& \_v);

Vector<T> operator -(const T& \_v);

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

Vector<T> operator /(const T& \_v);

Vector<T>& operator =(const Vector<T>& \_v);

T& operator[] (const int index);

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

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

Vector operator +(const Vector& \_v);

Vector operator -(const Vector& \_v);

Vector operator \*(const Vector& \_v);

Vector<T>& operator ++();

Vector<T>& operator --();

Vector<T>& operator +=(Vector<T>& \_v);

Vector<T>& operator -=(Vector<T>& \_v);

template <class T1>

friend ostream& operator<< (ostream& ostr, const Vector<T1> &A);

template <class T1>

friend istream& operator >> (istream& istr, Vector<T1> &A);

int Length();

int GetInd();

};

template <class T1>

ostream& operator<< (ostream& ostr, const Vector<T1> &A) {

for (int i = 0; i < A.length; i++) {

ostr << A.x[i] << endl;

}

return ostr;

}

template <class T1>

istream& operator >> (istream& istr, Vector<T1> &A) {

for (int i = 0; i < A.length; i++) {

istr >> A.x[i];

}

return istr;

}

#define MIN(a,b)(a>b?b:a)

#define MAX(a,b)(a>b?a:b)

template <class T>

Vector<T>::Vector(int \_v, int \_ind)

{

if (\_v < 0||\_v > MAX\_VECTOR\_SIZE||\_ind<0)

throw new exception();

length = \_v;

ind = \_ind;

x = new T[length];

}

template <class T>

Vector<T>::Vector(const Vector<T>& \_v)

{

length = \_v.length;

ind = \_v.ind;

x = new T[length];

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

x[i] = \_v.x[i];

}

template <class T>

Vector<T>::~Vector()

{

length = 0;

if (x != NULL)

delete [] x;

x = 0;

ind = 0;

}

template<class T>

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

{

if (length != \_v.length || ind != \_v.ind)

return false;

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

if (x[i] != \_v.x[i])

return false;

return true;

}

template<class T>

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

{

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

if (x[i] == \_v.x[i])

return false;

return true;

}

template <class T>

Vector<T> Vector<T>::operator +(const T& \_v)

{

Vector<T> res;

res.length = length;

res.x = new T[res.length];

for (int i = 0; i < res.length; i++)

{

res.x[i] = x[i] + \_v;

}

return res;

}

template <class T>

Vector<T> Vector<T>::operator -(const T& \_v)

{

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

{

x[i] = x[i] - \_v;

}

return \*this;

}

template <class T>

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

{

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

{

x[i] = x[i] \* \_v;

}

return \*this;

}

template <class T>

Vector<T> Vector<T>::operator /(const T& \_v)

{

Vector<T> res;

res.length = MIN(length, \_v.length);

res.x = new T [res.length];

for (int i = 0; i < res.length; i++)

{

res.x[i] = x[i] / \_v.x[i];

}

return res;

}

template <class T>

Vector<T>& Vector<T>::operator =(const Vector<T>& \_v)

{

if (this == &\_v)

return \*this;

if (length != \_v.length)

{

delete[] x;

x = new T[\_v.length];

}

length = \_v.length;

ind = \_v.ind;

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

x[i] = \_v.x[i];

return \*this;

}

template <class T>

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

{

if ((index - ind) < 0 || (index - ind) >= length)

throw exception();

return x[index - ind];

}

template <class T>

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

{

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

x[i]++;

return \*this;

}

template <class T>

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

{

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

x[i]--;

return \*this;

}

template <class T>

Vector<T>& Vector<T>::operator +=(Vector<T>& \_v)

{

length = MIN(length, \_v.length);

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

{

x[i] += \_v.x[i];

}

return \*this;

}

template <class T>

Vector<T>& Vector<T>::operator -=(Vector<T>& \_v)

{

length = MIN(length, \_v.length);

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

x[i] -= \_v.x[i];

return \*this;

}

template <class T>

Vector<T> Vector<T>::operator+(const Vector& \_v)

{

if (\_v.length != length)

throw new exception;

Vector<T> res;

res.length = MIN(length, \_v.length);

res.x = new T[res.length];

for (int i = 0; i < res.length; i++)

res.x[i] = x[i] + \_v.x[i];

return res;

}

template <class T>

Vector<T> Vector<T>::operator-(const Vector& \_v)

{

if (\_v.length != length)

throw new exception;

Vector<T> res;

res.length = MIN(length, \_v.length);

res.x = new T[res.length];

for (int i = 0; i < res.length; i++)

res.x[i] = x[i] - \_v.x[i];

return res;

}

template <class T>

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

{

if (\_v.length != length)

throw new exception;

Vector<T> res;

res.length = MIN(length, \_v.length);

res.x = new T[res.length];

for (int i = 0; i < res.length; i++)

res.x[i] = x[i] \* \_v.x[i];

return res;

}

template <class T>

int Vector<T>::Length()

{

return length;

}

template<class T>

inline int Vector<T>::GetInd()

{

return ind;

}

#endif

Matrix.h

#include "MyVector.h"

const int MAX\_SIZE = 10000;

template <class T>

class TMatrix : public Vector<Vector<T> >

{

public:

TMatrix(int \_size=0);

TMatrix(const TMatrix<T>& A);

TMatrix(const Vector<Vector<T> >& A);

~TMatrix();

TMatrix<T>& operator = (const TMatrix<T>& A);

TMatrix<T> operator + (const TMatrix<T>& A);

TMatrix<T> operator - (const TMatrix<T>& A);

TMatrix<T> operator\*(TMatrix<T>& A);

bool operator == (const TMatrix<T>& A) const;

bool operator != (const TMatrix<T>& A) const;

friend ostream& operator<< (ostream& ostr, const TMatrix& A)

{

for (int i = 0; i < A.length; i++)

ostr << A.x[i] << endl;

return ostr;

}

friend istream& operator>> (istream& istr, TMatrix& A)

{

for (int i = 0; i < A.Length; i++)

istr >> A.x[i];

return istr;

}

};

template<class T>

inline TMatrix<T>::TMatrix(int \_size) : Vector<Vector<T> >(\_size)

{

if (\_size < 0 || \_size > MAX\_SIZE)

throw new exception;

for (int i = 0; i < \_size; i++)

this->x[i] = Vector<T>(\_size-i);

}

template<class T>

inline TMatrix<T>::TMatrix(const TMatrix<T>& A) : Vector<Vector<T> >(A)

{

}

template<class T>

inline TMatrix<T>::TMatrix(const Vector<Vector<T> >& A) : Vector<Vector<T> >(A)

{

}

template<class T>

inline TMatrix<T>::~TMatrix()

{

}

template<class T>

inline TMatrix<T>& TMatrix<T>::operator=(const TMatrix<T>& A)

{

if (this != &A)

{

if (this->length != A.length)

{

if(this->x!=NULL)

delete[] this->x;

this->length = A.length;

this->x = new Vector<T>[A.length];

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

this->x[i] = A.x[i];

}

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

this->x[i] = A.x[i];

}

return \*this;

}

template<class T>

inline TMatrix<T> TMatrix<T>::operator+(const TMatrix<T>& A)

{

if (this->length != A.length)

throw "Error";

TMatrix<T> tmp(\*this);

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

tmp.x[i] += A.x[i];

return tmp;

}

template<class T>

inline TMatrix<T> TMatrix<T>::operator-(const TMatrix<T>& A)

{

if (this->length != A.length)

throw new exception;

TMatrix<T> tmp(\*this);

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

tmp.x[i] = tmp.x[i] - A.x[i];

return tmp;

}

template<class T>

inline TMatrix<T> TMatrix<T>::operator\*(TMatrix<T>& A)

{

TMatrix<T> tmp(\*this);

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

tmp.x[i] = tmp.x[i] \* A.x[i];

return tmp;

}

template<class T>

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

{

if (this->length != A.length)

return false;

for (int i = 0; i < A.length; i++)

if (this->x[i] != A.x[i])

return false;

return true;

}

template<class T>

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

{

for (int i = 0; i < A.length; i++)

if (this->x[i] == A.x[i])

return false;

return true;

}