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

Структура хранения матриц специального вида

Выполнил:

студент ф-та ИИТММ гр. 381808-2

Подскребко Н.В.

Нижний Новгород

2019 г.

Содержание

Введение…………………………………………………………………………………………3

Постановка Задачи……………………………………………………………………………...4

Руководство Пользователя……………………………………………………………………..5

Руководство Программиста……………………………………………………………………6

* Описание Структуры Программы……………………………………………………..6
* Описание Структур Данных…………………………………………………………...7
* Описание алгоритмов………………………………………………………………….10

Заключение…………………………………………………………………………………….11

Приложение…………………………………………………………………………………....12

Тесты…………………………………………………………………………………………...18

**Введение:**

Понятие **Матрица** в европейской науке было введено в середине XIX века. Но что же она представляет в понятном для нас научном объяснении?

**Матрица** есть структура данных, хранящая набор значений, идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона. Данная структура данных примечательна тем, что она предоставляет быстрый доступ к данным, при том не занимая много места (ниже диагонали = 0).

// <https://ru.wikipedia.org/wiki/Массив_(тип_данных)>

Матричные обозначения широко распространены в современной математике и её приложениях. Матрица – полезный аппарат для исследования многих задач теоретической и прикладной математики.

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.

**Цель Работы:**

**Цель данной работы** — разработка структуры данных для хранения множеств, поддерживающую эффективное хранение матриц специального вида (верхнетреугольных/верхних треугольных) и выполнение основных операций над ними:

* сложение/вычитание;
* копирование;
* сравнение.

В процессе выполнения лабораторной работы требуется использовать систему контроля версий Git и фрэймворк для разработки автоматических тестов Google Test.

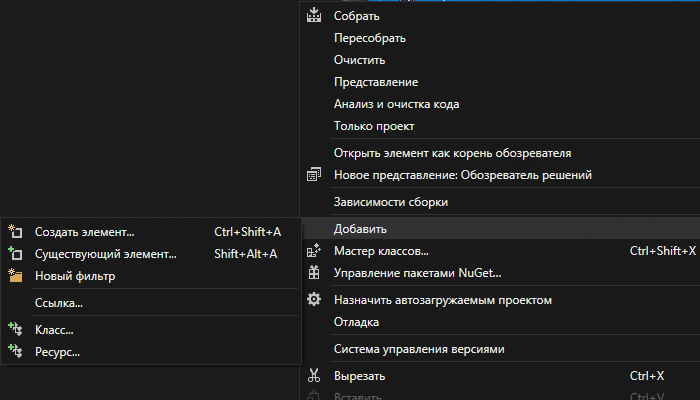
Выполнение работы предполагает решение следующих задач:

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.

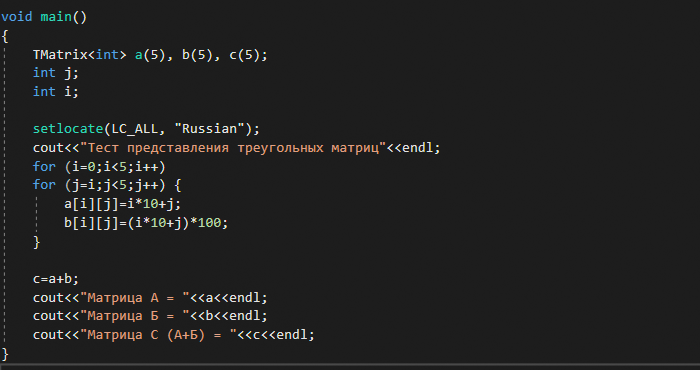
**Руководство Пользователя:**

В ходе лабораторной работы был получен заголовочный файл **utmatrix.h**, для его использования необходимо:

* В свойство вашего проекта добавить существующий элемент, выбираем **utmatrix.h**
* В нужном файле – подключаем заголовочный файл **#include <utmatrix.h**



* Теперь можно пользоваться классом **TSet**, а также его методами.



**Руководство Программиста:**

**Описание структуры программы:**

Статическая Библиотека **Set**:

* В ней содержатся заголовочные файлы – **tbitfield.h** и **TSet.h**. В них, в свою очередь, определены интерфейсы классов битового поля **TBitField** и множеств **TSet**. Файлы их реализации – **TbitField.cpp**, **TSet.cpp**.
* Модуль **Test\_set** содержит для каждого класса **Tmatrix** и **TVector** набор тестов, реализованный в файлах **test\_tamtrix.cpp** и **test\_tvector.cpp** с помощью использования фреймворка Google Test.
* Модуль **sample\_matrix** содержит файл **sample\_matrix.cpp**, в котором реализован пример использования класса для отображения верхней треугольной матрицы.

**Описание структур данных:**

Класс **TMatrix** реализован путем наследования класса **TVector<Tbector<Type>**

Поля класса:

* **Int Size** – размер вектора.
* **Int StartIndex** – индекс первого элемента вектора (матрицы)
* **ValType \*pvector** массив, хранящий элементы вектора (матрицы)

Класс верхней треугольная матрица представлена вектором векторов шаблонного типа, элементы которой ниже главной диагонали отсутствуют, но подразумевается, что они равны 0.

**Векторы**:

* Класс **TVector** – шаблонный вектор:

Вектор - одномерный массив проиндексированных элементов. Массив pVector рассматривается как последовательность элементов, расположенных непрерывно в памяти. Доступ к каждому элементу осуществляется через операцию индексации ([]).

Элементы, объявленные со спецификатором **private:**

**int Size** – длина вектора

**int StartIndex** – индекс первого элемента

**int \* pVector** – память для представления вектора

Со спецификатором доступа **public**:

**TVector (int s=10,int si=0)** – конструктор инициализации, одновременно конструктор по умолчанию. Принимает длину вектора и индекс первого элемента.

**TVector (const TBitField &bf)** – конструктор копирования.

**~ TVector ()** – деструктор

**Методы**

**int GetSize() const** – возвращает длину вектора

**int GetStartIndex()** – возвращает индекс первого элемента

**ValType& operator[](int pos)** – предоставляет доступ к элементу [pos-si]

**int GetBit(const int n) const** – возвращает значение бита n

**Bool operator==(const TVector &v) const** – принимает ссылку на объект класса **TVector**, проверяет на равенство два вектора. Если они идентичны, то возвращает 1, иначе 0

**Bool operator!=(const TVector &v) const** – принимает ссылку на объект класса **TVector**, проверяет на равенство два вектора. Если они не идентичны, то возвращает 1, иначе 0

**TVector & operator=(const TVector &м)** – принимает ссылку на объект класса **TVector**, присваивает полученный вектор к тому, которым был вызван оператор.

**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 элементу вектора на значение val

**TVector & operator+(const ValType &val)** – прибавляет к каждому элементу вектора значение val

**TVector & operator+(const TVector &v)** – если размеры векторов равны, то поэлементно складывает два вектора

**TVector & operator-(const TVector &v)** – если размеры векторов равны, то поэлементно вычитает из нашего вектора val

**TVector & operator\*(const TVector &v)** – если размеры векторов равны, то вычисляет скалярное произведение векторов

Дружественные функции:

**friend istream &operator>>(istream &istr, TVector &v)** – ввод вектора с консоли.

**friend ostream &operator<<(ostream &ostr, const TVector &v)** – вывод вектора в консоль.

* Класс **TMatrix** – матрица:

Элементы, объявленные со спецификатором **private:**

Поля наследованы от шаблонного класса **Vector<Vector<ValType>**

Со спецификатором доступа **public:**

**TMatrix(int s)** – конструктор-инициализатор. Принимает размер матрицы

**TMatrix(const TMatrix &s)** – конструктор копирования.

**TMatrix(const TVector<TVector<ValType>> &mt)** – конструктор преобразования типа. Принимает ссылку на объект с типом **TVector<TVector<ValType>>**

**Bool operator==(const TMatrix &v) const** – принимает ссылку на объект класса **TMatrix**, проверяет на равенство два вектора. Если они идентичны, то возвращает 1, иначе 0

**Bool operator!=(const TMatrix &v) const** – принимает ссылку на объект класса **TMatrix**, проверяет на равенство два вектора. Если они не идентичны, то возвращает 1, иначе 0

**TMatrix & operator=(const TMatrix &м)** – принимает ссылку на объект класса **TMatrix**, присваивает полученный вектор к тому, которым был вызван оператор

**TMatrix & operator+(const TMatrix &v)** – если размеры векторов равны, то построчно складывает два вектора, иначе пересоздает объект и дальше сложение

**TMatrix & operator-(const TMatrix &v)** – если размеры векторов равны, то построчно вычитает из нашей матрицу принимаемую, иначе пересоздает объект и дальше вычитание

Дружественные функции:

**friend istream &operator>>(istream &istr, TMatrix &v)** – ввод матрицы с консоли

**friend ostream &operator<<(ostream &ostr, const TMatrix &v)** – вывод матрицы в консоль

**Описание алгоритмов:**

Создание **верхней треугольной матрицы:**

Сходство в задании структуры данных **Vector**, как набора элементов, связанных отношением следования, и структуры данных **Matrix**, как набора векторов, связанных отношением следования.

Этот факт позволяет создать объект матрица, построенного на векторе векторов. А именно все векторы будут размером **n-i, где i от 0 до n-1**.

Алгоритм: в цикле от 0 до длины матрицы присваивать элементам массива **pVector[i]** присваиваем локально созданный объект класса, а именно:

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

так как нам нужна именно верхняя треугольная матрица, то размер с каждым шагом уменьшается.

**Заключение:**

* По итогу – была разработана структура данных для хранения множеств, поддерживающую эффективное хранение матриц специального вида (верхнетреугольных/верхних треугольных) и выполнение основных операций над ними.
* Данная структура позволяет экономить памяти, если есть возможность представить информация в виде матрицы, где элементы ниже главной диагонали равны 0, или нужная матрица уже в данном виде.
* Применение данной структуры экономит **\*4 байт** памяти!

**Приложение:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// utmatrix.h - Copyright (c) Гергель В.П. 07.05.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (21.04.2015)

//

// Верхнетреугольная матрица - реализация на основе шаблона вектора

#ifndef \_\_TMATRIX\_H\_\_

#define \_\_TMATRIX\_H\_\_

#include <iostream>

using namespace std;

const int MAX\_VECTOR\_SIZE = 100000000;

const int MAX\_MATRIX\_SIZE = 10000;

// Шаблон вектора

template <class ValType>

class TVector

{

protected:

ValType\* pVector;

int Size; // размер вектора

int StartIndex; // индекс первого элемента вектора

public:

TVector(int s = 10, int si = 0); // одновременно конструктор по умолчанию

// и конструктор с параметрами

TVector(const TVector& v); // конструктор копирования

~TVector();

int GetSize() { return Size; } // размер вектора

int GetStartIndex() { return StartIndex; } // индекс первого элемента

ValType& operator[](int pos); // доступ

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

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

TVector& operator=(const TVector& v); // присваивание

// скалярные операции

TVector operator+(const ValType& val); // прибавить скаляр

TVector operator-(const ValType& val); // вычесть скаляр

TVector operator\*(const ValType& val); // умножить на скаляр

// векторные операции

TVector operator+(const TVector& v); // сложение

TVector operator-(const TVector& v); // вычитание

ValType operator\*(const TVector& v); // скалярное произведение

// 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.StartIndex; i++) {

out << "0" << ' ';

}

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

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

return out;

}

};

template <class ValType>

TVector<ValType>::TVector(int s, int si)//:StartIndex(si),Size(s)

{

if ((s >= 0) && (s <= MAX\_VECTOR\_SIZE))

{

if ((si > -1) && (si < MAX\_VECTOR\_SIZE))

{

pVector = new ValType[s];

Size = s;

StartIndex = si;

}

else

throw (si);

}

else

throw(s);

}

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

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)

{

delete[] pVector;

pVector = nullptr;

}

}

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

template <class ValType> // доступ

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

{

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

throw ("Position is not correct");

return pVector[pos - StartIndex];

}

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

template <class ValType> // сравнение

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

{

if (&v == this)

return true;

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 ValType> // сравнение

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

{

if (Size != v.Size && 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)

return \*this;

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> temp\_var(Size, StartIndex);

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

{

temp\_var[i] = pVector[i] + val;

}

return temp\_var;

}

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

template <class ValType> // вычесть скаляр

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

{

TVector<ValType> temp\_var(Size, StartIndex);

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

temp\_var[i] = pVector[i] - val;

}

return temp\_var;

}

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

template <class ValType> // умножить на скаляр

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

{

TVector<ValType> temp\_var(Size, StartIndex);

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

temp\_var[i] = pVector[i] \* val;

}

return temp\_var;

}

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

template <class ValType> // сложение

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

{

if (Size != v.Size)

throw("Not equal sizes");

TVector<ValType> temp\_var(Size, StartIndex);

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

temp\_var.pVector[i] = pVector[i] + v.pVector[i];

}

return temp\_var;

}

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

template <class ValType> // вычитание

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

{

if (Size != v.Size)

throw("Not equal sizes");

TVector<ValType> temp\_var(Size, StartIndex);

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

temp\_var.pVector[i] = pVector[i] - v.pVector[i];

}

return temp\_var;

}

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

template <class ValType> // скалярное произведение

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

{

if (Size != v.Size)

throw("Not equal sizes");

ValType sum = 0;

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

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

return sum;

}

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

// Верхнетреугольная матрица

template <class ValType>

class TMatrix : public TVector<TVector<ValType> >

{

public:

TMatrix(int s = 10);

TMatrix(const TMatrix& mt); // копирование

TMatrix(const TVector<TVector<ValType> >& mt); // преобразование типа

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

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

TMatrix& operator= (const TMatrix& mt); // присваивание

TMatrix operator+ (const TMatrix& mt); // сложение

TMatrix operator- (const TMatrix& mt); // вычитание

// 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)

{

if (s > MAX\_MATRIX\_SIZE || s <= 0)

throw "Error";

Size = s;

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

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

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

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)

return \*this;

if (mt.Size != Size) {

delete[] pVector;

Size = mt.Size;

pVector = new TVector<ValType>[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)

{

if (Size != mt.Size || StartIndex != mt.StartIndex)

throw "Error";

TMatrix<ValType> a(Size);

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

{

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

}

return a;

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

template <class ValType> // вычитание

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

{

if (Size != mt.Size || StartIndex != mt.StartIndex)

throw "Error";

//return TVector<TVector<ValType>>::operator-(mt);

TMatrix<ValType> a(Size);

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

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

}

return a;

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

// TVector О3 Л2 П4 С6

// TMatrix О2 Л2 П3 С3

#endif

**Тесты:**

