ФЕДЕРАЛЬНОЕ АГЕНСТВО ЖЕЛЕЗНОДОРОЖНОГО ТРАНСПОРТА

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПУТЕЙ СООБЩЕНИЯ Императора Александра I»

Кафедра «Информационные и вычислительные системы»

Дисциплина «Программирование С++»

**ОТЧЁТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7**

ВАРИАНТ 19

|  |  |
| --- | --- |
| Выполнил студент  Факультет: АИТ  Группа: ИВБ-211 | Шефнер А. |
| Проверил: | Проузин О.В. |

**Санкт-Петербург**

**2023**

Оценочный лист результатов ЛР № 7

Ф.И.О. студента \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Шефнер Альберт\_\_\_\_\_

Группа \_\_\_\_\_\_\_\_\_\_\_\_\_ИВБ-211\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№**  **п/п** | **Материалы необходимые для оценки знаний, умений**  **и навыков** | **Показатель**  **оценивания** | **Критерии**  **Оценивания** | **Шкала оценивания** | **Оценка** |
| 1 | Лабораторная работа№ | Соответствие методике выполнения | Соответствует | 7 |  |
| Не соответствует | 0 |
| Срок выполнения | Выполнена в срок | 2 |  |
| Выполнена с опозданием на 2 недели | 0 |
| оформление | Соответствует требованиям | 1  0 |  |
| Не соответствует |  |
|  | **ИТОГО количество баллов** |  |  | 10 |  |

Доцент кафедры

«Информационные и вычислительные

системы» Проурзин О.В. «\_\_» \_\_\_\_\_\_\_\_\_\_2023 г.

**Цели работы.**

* Использовать шаблоны классов.

**Задание.**

1. Создать класс-целое, который описывает массив объектов авторского класса из Лабораторной работы № 4.

* Массив указателей на объекты авторского класса из Лабораторной работы № 4 – открытый член-данное;
* Фактическое количество объектов в массиве – закрытый член-данное.
* Конструктор с умолчанием (устанавливает нулевое значение фактического количества объектов в массиве).
* Поиск объекта в массиве по значению ключа.
* Добавление объекта в массив (объекты с одинаковым ключом недопустимы).
* Удаление объекта из массива по ключу.

1. Добавить метод проверки ключа в класс из Лабораторной работы № 4.
2. Написать тестовую функцию main ( ), в которой создать объект класса-целого, а затем осуществить проверку работоспособности следующих его методов:

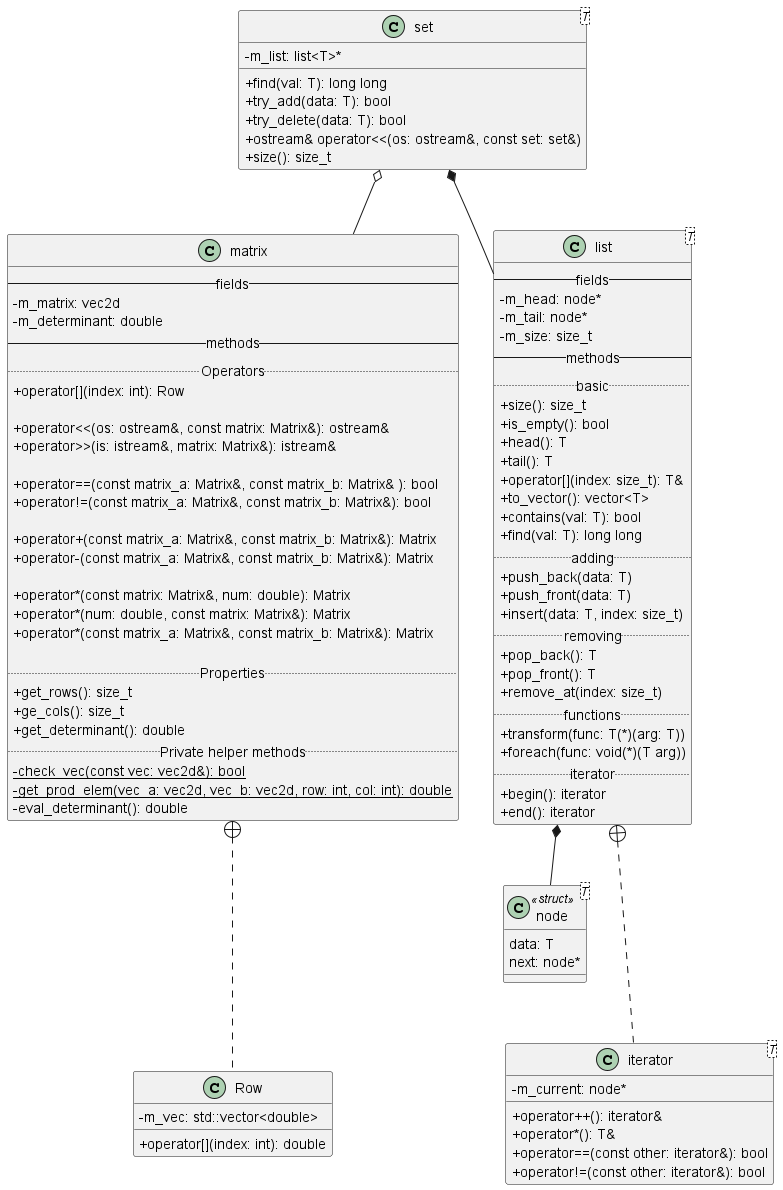
* поиск объекта в массиве по ключу;
* добавление объекта в массив;
* удаление объекта из массива;
* просмотр состояния массива объектов.

**Используемые средства**

В качестве интегрированной среды разработки использовалась JetBrains CLion.

Для работы в консоли с потоками ввода-вывода использовалась стандартная библиотека <iostream>.

**UML-диаграмма программы:**



**Код программы**

**main.cpp**

#include <iostream>

#include <memory>

#include <fstream>

#include "list.h"

#include "matrix.h"

#include "set.h"

int main(int argc, char\* argv[])

{

std::ifstream fin("matrices.txt");

std::unique\_ptr<list<matrix>> matrix\_list = std::make\_unique<list<matrix>>();

std::unique\_ptr<set<matrix>> matrix\_set = std::make\_unique<set<matrix>>();

matrix tmp;

while(!fin.eof())

{

fin >> tmp;

matrix\_list->push\_back(tmp);

matrix\_set->try\_add(tmp);

}

std::cout << "List of matrices:\n";

std::cout << "count: " << matrix\_list->size() << '\n';

for(auto& m : \*matrix\_list)

{

std::cout << m << "\n";

}

std::cout << '\n' << "Set of matrices:\n" << "count: " << matrix\_set->size() << '\n';

std::cout << \*matrix\_set << std::endl;

system("pause");

}

**set.h**

#pragma once

#include <iostream>

#include <memory>

#include "list.h"

template <typename T>

class set

{

public:

set() : m\_list(std::make\_unique<list<T>>()) {}

set(const set& \_set) : m\_list(std::make\_unique<list<T>>(\_set.m\_list)) {}

// поиск объекта в массиве по ключ

long long find(T val) const { return m\_list->find(val); }

// Добавление объедка в массив

bool try\_add(T data)

{

if(m\_list->contains(data)) return false;

m\_list->push\_back(data);

return false;

}

// Удаление объедка из массива

bool try\_delete(T data)

{

auto index = m\_list->find(data);

if(index == -1) return false;

m\_list->remove\_at(index);

return true;

}

// просмотр состояния массива объектов.

friend std::ostream& operator<<(std::ostream& os, const set& set)

{

for(auto x: \*set.m\_list) os << x << "\n";

os << '\n';

return os;

}

size\_t size() {return m\_list->size(); }

private:

std::unique\_ptr<list<T>> m\_list;

};

**list.h**

#pragma once

#include <memory>

#include <vector>

/\*\*

\* \brief

\* Singly linked list for data

\* \tparam T the type of the data

\*/

template <typename T>

class list

{

public:

list() = default;

list(const list& cpy)

{

auto tmp = cpy.m\_head;

while (tmp != nullptr)

{

push\_back(tmp->data);

tmp = tmp->next;

}

}

~list() = default;

/\*\*

\* \brief

\* Returns the count of elements currently in the list.

\*/

size\_t size() const { return m\_size; }

/\*\*

\* \brief

\* Checks whenever the list is empty or not.

\* \return True if the list is empty or false if the list is not empty.

\*/

bool is\_empty() const { return m\_size == 0; }

/\*\*

\* \brief

\* Get the data of the lists head.

\* \return The data of type T of the lists head.

\*/

T head() const;

/\*\*

\* \brief

\* Get the data of the lists tail.

\* \return The data of type T of the lists tail.

\*/

T tail() const;

/\*\*

\* \brief

\* Returns a data from the list at index.

\* \param index the index of the element in order from head to tail.

\* \return The data of type T at specified index.

\*/

T& operator[](size\_t index);

/\*\*

\* \brief

\* Converts the list to std::vector<T>.

\* \return std::vector<T>, containing all elements of

\* the list with the same template type T.

\*/

std::vector<T> to\_vector();

/\*\*

\* \brief

\* Pushes the data at the end of the list.

\* \param data a data of the type T.

\*/

void push\_back(T data);

/\*\*

\* \brief

\* Pushes the data at the start of the list.

\* \param data a data of the type T.

\*/

void push\_front(T data);

/\*\*

\* \brief Inserts data in the list at specified index.

\* \details

\* If the index is less than 0 or greater or equal

\* to the lists size than exception will be thrown.

\* \param data a data of the type T.

\* \param index an index at which the new element will be in the list.

\*/

void insert(T data, size\_t index);

/\*\*

\* \brief

\* Removes an elements at the end of the list and returns its value.

\* \return data of the type T.

\*/

T pop\_back();

/\*\*

\* \brief

\* Removes an elements at the start of the list and returns its value.

\* \return data of the type T.

\*/

T pop\_front();

/\*\*

\* \brief

\* Removes an element from the list at the specified index.

\* \param index the index of the element to remove.

\*/

void remove\_at(size\_t index);

/\*\*

\* \brief

\* Applies a function to all elements of the list.

\* \param func The function with one parameter of the type T

\* which returns value of the type T.

\*/

void transform(T (\*func)(T arg));

/\*\*

\* \brief

\* Calls a function with each element of the list as a parameter.

\* \param func The function with one parameter of the type T.

\*/

void foreach(void (\*func)(T arg));

/\*\*

\* \brief

\* Checks if the list contains the value.

\* \param val The value for check.

\* \return True if the value is found in the list, false otherwise.

\*/

bool contains(T val) const;

/\*\*

\* \brief

\* Finds an index of the first element of the list, equals to value.

\* \param val The value to find.

\* \return The index of the found element if the list

\* contains the value, -1 otherwise.

\*/

long long find(T val) const;

private:

struct node

{

explicit node(T \_data) : data(\_data){}

T data;

std::shared\_ptr<node> next;

};

std::shared\_ptr<node> m\_head = nullptr;

std::shared\_ptr<node> m\_tail = nullptr;

size\_t m\_size = 0;

//Iteration over list

public:

class iterator

{

public:

explicit iterator(const std::shared\_ptr<node>& current) : m\_current(current){}

iterator& operator++() { m\_current = m\_current->next; return \*this; }

T& operator\*() const { return m\_current->data; }

T& operator\*() { return m\_current->data; }

bool operator==(const iterator& other) const { return m\_current == other.m\_current; }

bool operator!=(const iterator& other) const { return m\_current != other.m\_current; }

private:

std::shared\_ptr<node> m\_current;

};

iterator begin() { return iterator(m\_head); }

iterator begin() const { return iterator(m\_head); }

iterator end() { return iterator(nullptr); }

iterator end() const { return iterator(nullptr); }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \* \*

// \* Implementation. \*

// \* \*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <typename T>

T list<T>::head() const

{

if(m\_size == 0) throw -1;

return m\_head->data;

}

template <typename T>

T list<T>::tail() const

{

if(m\_size == 0) throw -1;

return m\_tail->data;

}

template <typename T>

T& list<T>::operator[](const size\_t index)

{

if(index >= m\_size || index < 0) throw -1;

std::shared\_ptr<node> tmp = m\_head;

for(size\_t i = 0; i < index; i++)

{

tmp = tmp->next;

if(tmp == nullptr) throw -1;

}

return tmp->data;

}

template <typename T>

std::vector<T> list<T>::to\_vector()

{

std::vector<T> vec;

std::shared\_ptr<node> tmp = m\_head;

for(size\_t i = 0; i < m\_size; i++)

{

vec.push\_back(tmp->data);

tmp = tmp->next;

}

return vec;

}

template <typename T>

void list<T>::push\_back(T data)

{

std::shared\_ptr<node> new\_node = std::make\_shared<node>(data);

if(m\_size == 0) m\_head = new\_node;

else m\_tail->next = new\_node;

m\_tail = new\_node;

m\_size++;

}

template <typename T>

void list<T>::push\_front(T data)

{

if(m\_size == 0)

{

push\_back(data);

return;

}

auto new\_node = std::make\_shared<node>(data);

new\_node->next = m\_head;

m\_head = new\_node;

if(m\_size == 1) m\_tail = m\_head->next;

m\_size++;

}

template <typename T>

void list<T>::insert(T data, const size\_t index)

{

if(index > m\_size || index < 0) throw -1;

if(index == m\_size)

{

push\_back(data);

return;

}

if(index == 0)

{

push\_front(data);

return;

}

auto tmp = m\_head;

auto new\_node = std::make\_shared<node>(data);

for(size\_t i = 0; i < index - 1; i++)

{

tmp = tmp->next;

}

new\_node->next = tmp->next;

tmp->next = new\_node;

m\_size++;

}

template <typename T>

T list<T>::pop\_back()

{

if(m\_size== 0) throw -1;

T return\_data = m\_tail->data;

if(m\_size == 1)

{

m\_tail = nullptr;

m\_head = nullptr;

m\_size--;

return return\_data;

}

auto tmp = m\_head;

for(size\_t i = 0; i < m\_size - 2; i++)

{

tmp = tmp->next;

}

m\_tail = tmp;

m\_size--;

return return\_data;

}

template <typename T>

T list<T>::pop\_front()

{

if(m\_size == 0) throw -1;

if(m\_size == 1) return pop\_back();

T data = m\_head->data;

m\_head = m\_head->next;

m\_size--;

return data;

}

template <typename T>

void list<T>::remove\_at(size\_t index)

{

if(m\_size == 0 || index < 0) throw -1;

if(index == 0)

{

pop\_front();

return;

}

if(index == m\_size - 1)

{

pop\_back();

return;

}

if(m\_size == 1)

{

m\_head = nullptr;

m\_tail = nullptr;

return;

}

auto tmp = m\_head;

for(size\_t i = 0; i < index - 1; i++)

{

tmp = tmp->next;

}

tmp->next = tmp->next->next;

m\_size--;

}

template <typename T>

void list<T>::transform(T(\* func)(T arg))

{

auto tmp = m\_head;

for(size\_t i = 0; i < m\_size; i++)

{

tmp->data = func(tmp->data);

tmp = tmp->next;

}

}

template <typename T>

void list<T>::foreach(void(\* func)(T arg))

{

auto tmp = m\_head;

for(size\_t i = 0; i < m\_size; i++)

{

func(tmp->data);

tmp = tmp->next;

}

}

template <typename T>

bool list<T>::contains(T val) const

{

for(const auto el : \*this)

{

if(el == val) return true;

}

return false;

}

template <typename T>

long long list<T>::find(T val) const

{

int index = 0;

for(const auto el : \*this)

{

if(el == val) return index;

index++;

}

return -1;

}

**matrix.h**

#pragma once

#include <iostream>

#include <vector>

typedef std::unique\_ptr<std::vector<std::vector<double>>> vec2d\_ptr;

typedef std::vector<std::vector<double>> vec2d;

class matrix {

public:

matrix();

matrix(const vec2d& vec);

matrix(const matrix& matrix);

~matrix();

friend bool operator ==(const matrix& matrix\_a, const matrix& matrix\_b);

friend bool operator !=(const matrix& matrix\_a, const matrix& matrix\_b) { return !(matrix\_a == matrix\_b); }

friend matrix operator +(const matrix& matrix\_a, const matrix& matrix\_b);

friend matrix operator -(const matrix& matrix\_a, const matrix& matrix\_b);

friend matrix operator \*(const matrix& matrix\_a, double num);

friend matrix operator \*(double num, const matrix& matrix) { return matrix \* num; }

friend matrix operator \*(const matrix& matrix\_a, const matrix& matrix\_b);

friend std::ostream& operator<<(std::ostream& os, const matrix& m);

friend std::istream& operator>>(std::istream& is, matrix& m);

size\_t get\_rows() const { return m\_matrix->size(); }

size\_t get\_cols() const { return (\*m\_matrix)[0].size(); }

double get\_determinant();

class Row {

public:

explicit Row(std::vector<double>& vec) : m\_vec(vec) {}

double operator[](size\_t index) const;

private:

std::vector<double>& m\_vec;

};

Row operator[](size\_t index) const;

private:

vec2d\_ptr m\_matrix;

double m\_determinant;

static bool check\_vec(const vec2d& vec);

static double get\_prod\_elem(const vec2d& vec\_a, const vec2d& vec\_b, size\_t row, size\_t col);

double eval\_determ() const;

};

**Matrix.cpp**

#include "matrix.h"

#include <cmath>

#include "determinant.h"

#define DETERMINANT\_NOT\_CALCULATED "1"

matrix::matrix()

{

m\_matrix = std::make\_unique<vec2d>();

m\_determinant = std::nan(DETERMINANT\_NOT\_CALCULATED);

}

matrix::matrix(const vec2d& vec)

{

if (!check\_vec(vec))throw -1;

m\_matrix = std::make\_unique<vec2d>(vec);

m\_determinant = std::nan(DETERMINANT\_NOT\_CALCULATED);

}

matrix::matrix(const matrix& matrix)

{

m\_matrix = std::make\_unique<vec2d>();

for (size\_t i = 0; i < matrix.get\_rows(); i++)

{

std::vector<double> row;

for (size\_t j = 0; j < matrix.get\_cols(); j++)

{

row.push\_back((\*matrix.m\_matrix)[i][j]);

}

m\_matrix->push\_back(row);

}

m\_determinant = matrix.m\_determinant;

}

matrix::~matrix() = default;

std::ostream& operator<<(std::ostream& os, const matrix& matrix)

{

for (size\_t i = 0; i < matrix.get\_rows(); i++)

{

for (size\_t j = 0; j < matrix.get\_cols(); j++)

{

os << (\*matrix.m\_matrix)[i][j] << " ";

}

os << '\n';

}

return os;

}

std::istream& operator>>(std::istream& is, matrix& m)

{

m.m\_matrix = std::make\_unique<std::vector<std::vector<double>>>();

size\_t rows, cols;

is >> rows >> cols;

for(size\_t i = 0; i < rows; i++)

{

std::vector<double> row\_vec;

for(size\_t j = 0; j < cols; j++)

{

double tmp;

is >> tmp;

row\_vec.push\_back(tmp);

}

m.m\_matrix->push\_back(row\_vec);

}

return is;

}

bool matrix::check\_vec(const vec2d& vec)

{

if (vec.empty()) return false;

const size\_t rowSize = vec[0].size();

for (const auto& row : vec)

{

if (row.size() != rowSize) return false;

}

return true;

}

matrix::Row matrix::operator[](size\_t index) const

{

if (index >= get\_rows() || index < 0) throw -1;

return Row((\*m\_matrix)[index]);

}

double matrix::Row::operator[](size\_t index) const

{

if (index >= m\_vec.size() || index < 0) throw -1;

return m\_vec[index];

}

bool operator==(const matrix& matrix\_a, const matrix& matrix\_b)

{

if (matrix\_a.get\_rows() != matrix\_b.get\_rows() || matrix\_a.get\_cols() != matrix\_b.get\_cols()) return false;

for (size\_t i = 0; i < matrix\_a.get\_rows(); i++)

{

for (size\_t j = 0; j < matrix\_a.get\_cols(); j++)

{

if ((\*matrix\_a.m\_matrix)[i][j] != (\*matrix\_b.m\_matrix)[i][j]) return false;

}

}

return true;

}

matrix operator+(const matrix& matrix\_a, const matrix& matrix\_b)

{

if (matrix\_a.get\_rows() != matrix\_b.get\_rows() || matrix\_a.get\_cols() != matrix\_b.get\_cols())

throw -1;

vec2d newVec;

for (size\_t i = 0; i < matrix\_a.get\_rows(); i++)

{

std::vector<double> row;

for (size\_t j = 0; j < matrix\_a.get\_cols(); j++)

{

row.push\_back((\*matrix\_a.m\_matrix)[i][j] + (\*matrix\_b.m\_matrix)[i][j]);

}

newVec.push\_back(row);

}

return newVec;

}

matrix operator-(const matrix& matrix\_a, const matrix& matrix\_b)

{

if (matrix\_a.get\_rows() != matrix\_b.get\_rows() || matrix\_a.get\_cols() != matrix\_b.get\_cols())

throw -1;

vec2d newVec;

for (size\_t i = 0; i < matrix\_a.get\_rows(); i++)

{

std::vector<double> row;

for (size\_t j = 0; j < matrix\_a.get\_cols(); j++)

{

row.push\_back((\*matrix\_a.m\_matrix)[i][j] - (\*matrix\_b.m\_matrix)[i][j]);

}

newVec.push\_back(row);

}

return newVec;

}

matrix operator\*(const matrix& matrix\_a, double num)

{

vec2d newVec;

for (size\_t i = 0; i < matrix\_a.get\_rows(); i++)

{

std::vector<double> row;

for (size\_t j = 0; j < matrix\_a.get\_cols(); j++)

{

row.push\_back((\*matrix\_a.m\_matrix)[i][j] \* num);

}

newVec.push\_back(row);

}

return newVec;

}

matrix operator\*(const matrix& matrix\_a, const matrix& matrix\_b)

{

if (matrix\_a.get\_rows() != matrix\_b.get\_cols()) throw -1;

vec2d newVec;

for (size\_t i = 0; i < matrix\_a.get\_rows(); i++)

{

std::vector<double> row;

for (size\_t j = 0; j < matrix\_b.get\_rows(); j++)

{

row.push\_back(matrix::get\_prod\_elem(\*matrix\_a.m\_matrix, \*matrix\_b.m\_matrix, i, j));

}

newVec.push\_back(row);

}

return newVec;

}

double matrix::get\_prod\_elem(const vec2d& vec\_a, const vec2d& vec\_b, const size\_t row, const size\_t col)

{

double element = 0;

for (size\_t i = 0; i < vec\_b.size(); i++)

{

element += vec\_a[row][i] \* vec\_b[i][col];

}

return element;

}

double matrix::get\_determinant()

{

if (std::isnan(m\_determinant))

{

if (get\_rows() != get\_cols()) throw -1;

m\_determinant = eval\_determ();

}

return m\_determinant;

}

double matrix::eval\_determ() const

{

const auto matrix\_array = new double\*[get\_rows()];

for (size\_t i = 0; i < get\_rows(); i++)

{

matrix\_array[i] = new double[get\_rows()];

for (size\_t j = 0; j < get\_rows(); j++)

{

matrix\_array[i][j] = (\*m\_matrix)[i][j];

}

}

const double det = eval\_determinant(matrix\_array, get\_rows());

for (size\_t i = 0; i < get\_rows(); i++)

{

delete matrix\_array[i];

}

delete[] matrix\_array;

return det;

}

**determimant.h**

#pragma once

double eval\_determinant(double \*\*matrix, unsigned long long int size);

**determimant.cpp**

#include "determinant.h"

#define size\_t unsigned long long int

double \*\*sub\_matrix(double \*\*matrix, size\_t size, size\_t deletedRow, size\_t deletedCol);

double eval\_determinant(double \*\*matrix, unsigned long long int size) {

if(size == 1) return matrix[0][0];

if(size == 2) return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0];

double det = 0;

for(size\_t i = 0; i < size; i++) {

double\*\* subMatrix = sub\_matrix(matrix, size, 0, i);

det += eval\_determinant(subMatrix, size - 1) \* matrix[0][i] \* (i & 1 ? -1 : 1);

for(size\_t j = 0; j < size - 1; j++) delete[] subMatrix[j];

}

return det;

}

double \*\*sub\_matrix(double \*\*matrix, size\_t size, size\_t deletedRow, size\_t deletedCol) {

size -= 1;

double\*\* newmatrix = new double\*[size];

for(size\_t i = 0; i < size; i++) {

newmatrix[i] = new double[size];

}

for(size\_t row = 0, newRow = 0; row <= size; row++) {

if(row != deletedRow) {

for (size\_t col = 0, newCol = 0; col <= size; col++) {

if (col != deletedCol) {

newmatrix[newRow][newCol] = matrix[row][col] ;

newCol++;

}

}

newRow++;

}

}

return newmatrix;

}

**Тестовые примеры**



**Входной файл matrices.txt**

**2 2**

**5 6**

**7 4**

**2 2**

**5 6**

**1 2**

**2 2**

**9 8**

**5 6**

**2 2**

**5 6**

**1 2**

**2 2**

**5 5**

**5 6**

**3 3**

**9 8 5**

**3 4 3**

**6 7 4**

**2 2**

**5 5**

**5 6**