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 C++»

ОТЧЁТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7

ВАРИАНТ 19

Выполнил студент Шефнер А.
Факультет: АИТ
Группа: ИВБ-211

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.

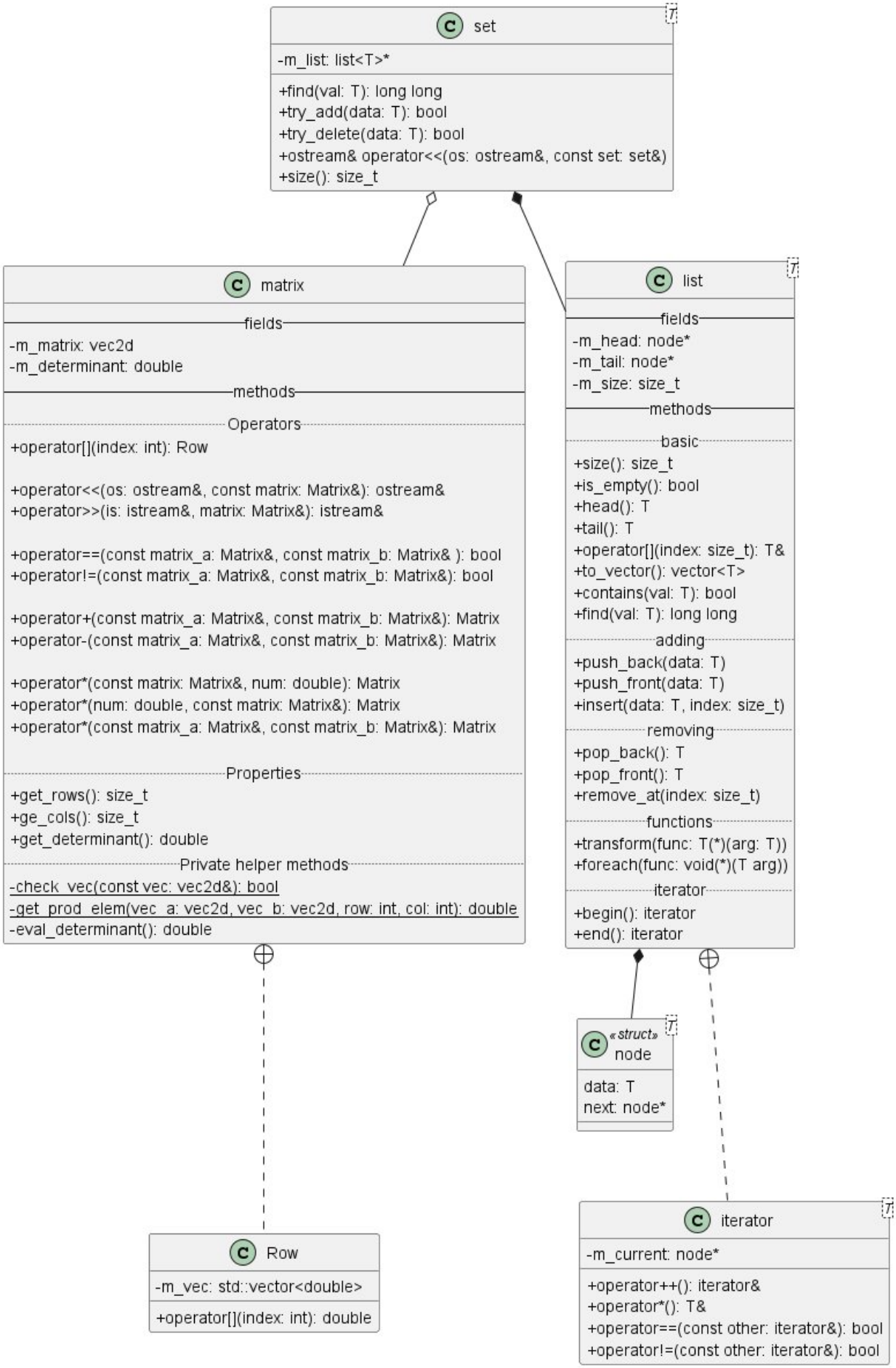
Задание.

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.
2. Добавить метод проверки ключа в класс из Лабораторной работы № 4.
3. 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

```
D:\University\2 term\Programming C++\Laboratory work 7\cpp_lab_7\x64\Debug>cpp_lab_7.exe
List of matrices:
count: 7
5 6
7 4

5 6
1 2

9 8
5 6

5 6
1 2

5 5
5 6

9 8 5
3 4 3
6 7 4

5 5
5 6


Set of matrices:
count: 5
5 6
7 4

5 6
1 2

9 8
5 6

5 5
5 6

9 8 5
3 4 3
6 7 4



Press any key to continue . . .
```

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
```