# МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Кемеровский государственный университет»

институт цифры кафедра цифровых технологий

# ОТЧЕТ

о выполнении

Лабораторной работы №5 **«ООП на С++: шаблоны»**

по курсу «Языки программирования»

студента 1 курса

Коник Ильи Николаевича

(ФИО полностью)

направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем

направленность (профиль) подготовки «Интеллектуальные информационные системы и анализ данных»

направление подготовки 02.03.02 Фундаментальная информатика и информационные технологии

направленность (профиль) подготовки «Информатика и компьютерные науки»

Преподаватель:

Дуванов И. О.

Кемерово 2025

**Тема: «ООП на С++: шаблоны»** *(6 баллов)*

## Задание

1. В соответствии с вариантами задания и приведённой ниже спецификацией реализуйте программу на языке C++.
2. Разработайте тестовые задания, и протестируйте программу одним из методов тестирования.

## Требования к оформлению программ:

* 1. **Содержание**. Программа должна делать только то, что предусмотрено заданием. Не нужно выполнять лишнюю работу.
  2. **Спецификация.** В преамбуле программы в комментариях указывать сведения:
     + Кто выполнил.
     + Что делает программа (кратко).
     + Что на входе, имена входных файлов указываются.
     + Что на выходе (что является результатов работы программы).

# Ввод и вывод

* + - Приглашение пользователю. (Например, сколько чисел, какого типа и через какой разделитель нужно вводить).
    - Контрольный вывод (все введенные данные выводить на экран, и только после этого выполнять необходимые вычисления.)
    - «Защита от дурака». Проверять вводимые данные на корректность. Например, если необходимо считать количество чего – то, то эта величина не может быть отрицательной и т.д.
  1. **Структура кода.** Набираемый код должен быть хорошо структурированным. Использовать:
     + Отступы.
     + Комментарии – поясняют решение программы.
     + Осмысленные названия переменных.
     + Пояснения о назначении переменных в комментариях (кроме счетчиков).

# Декомпозиция кода

* + - Функциональная
    - Объектно-ориентированная

# Многофайловые проекты

* + - Классы определять в отдельном h-файле, а все методы классов – в соответствующем cpp - файле. Созданный заголовочный файл подключать к проекту.
    - Методы класса реализовывать не в определении класса.
    - cpp-файл и соответствующий ему h-файл называть одинаково. В качестве

названия выбирать имя того класса, который определен в соответствующем модуле.

## Задания для самостоятельного выполнения:

***Задание 1.***

Реализуйте шаблонную функцию ***contains***, которая проверяет находится ли элемент (1 аргумент) в динамическом массиве (2 аргумент). Размер массива передается третьим аргументом.

Функция возвращает истину, если элемент найден, и ложь – если не найден. Продемонстрируйте работу функции на трех разных типах, включая строковый.

Array\_utils.cpp

#include "array\_utils.h"

#include <string>

template <typename T>

bool contains(const T& element, const T\* array, size\_t size) {

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

        if (array[i] == element) {

            return true;

        }

    }

    return false;

}

// Явные инстанциации шаблона для нужных типов

template bool contains<int>(const int&, const int\*, size\_t);

template bool contains<double>(const double&, const double\*, size\_t);

template bool contains<std::string>(const std::string&, const std::string\*, size\_t);

array\_utils.h

#ifndef ARRAY\_UTILS\_H

#define ARRAY\_UTILS\_H

#include <cstddef> // Для size\_t

// Шаблонная функция для проверки наличия элемента в массиве

template <typename T>

bool contains(const T& element, const T\* array, size\_t size);

#endif // ARRAY\_UTILS\_H

Main.cpp

#include "array\_utils.h"

#include <iostream>

#include <string>

int main() {

    // Пример 1: Массив целых чисел (int)

    int intArray[] = {10, 20, 30, 40, 50};

    size\_t intSize = sizeof(intArray) / sizeof(intArray[0]);

    int targetInt = 30;

    std::cout << "Is number " << targetInt << " contains in array? "

              << (contains(targetInt, intArray, intSize) ? "Yes" : "No") << std::endl;

    // Пример 2: Массив чисел с плавающей точкой (double)

    double doubleArray[] = {1.5, 2.7, 3.14, 5.0};

    size\_t doubleSize = sizeof(doubleArray) / sizeof(doubleArray[0]);

    double targetDouble = 2.71;

    std::cout << "Is number " << targetDouble << " contains in array? "

              << (contains(targetDouble, doubleArray, doubleSize) ? "Yes" : "No") << std::endl;

    // Пример 3: Массив строк (std::string)

    std::string stringArray[] = {"apple", "banana", "orange"};

    size\_t stringSize = sizeof(stringArray) / sizeof(stringArray[0]);

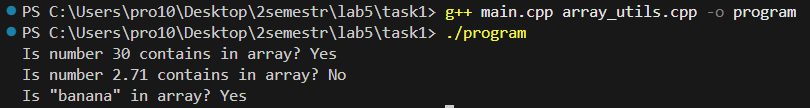
    std::string targetString = "banana";

    std::cout << "Is \"" << targetString << "\" in array? "

              << (contains(targetString, stringArray, stringSize) ? "Yes" : "No") << std::endl;

    return 0;

}



## Задание 2.

Реализуйте шаблонную функцию ***change***, которая меняет местами два элемента динамического массива. Она принимает четыре аргумента: два элемента, массив, размер массива. Если какого-то из элементов нет в массиве (для проверки используйте функцию contains из 1-ого задания), функция выводит сообщение об ошибке, завершается и возвращает ложь. Если обмен состоялся, функция выводит сообщение об успешном выполнении и возвращает истину. Продемонстрируйте работу функции на трех разных типах, включая строковый.

Array\_utils.cpp

#include "array\_utils.h"

#include <iostream>

template <typename T>

bool contains(const T& element, const T\* array, size\_t size) {

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

        if (array[i] == element) {

            return true;

        }

    }

    return false;

}

template <typename T>

bool change(const T& a, const T& b, T\* array, size\_t size) {

    if (!contains(a, array, size) || !contains(b, array, size)) {

        std::cerr << "Error: One or both elements are not present in the array!\n";

        return false;

    }

    size\_t indexA = 0, indexB = 0;

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

        if (array[i] == a) indexA = i;

        if (array[i] == b) indexB = i;

    }

    std::swap(array[indexA], array[indexB]);

    std::cout << "Elements \"" << a << "\" and \"" << b << "\" were successfully swapped.\n";

    return true;

}

template <typename T>

void printArray(const T\* array, size\_t size) {

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

        std::cout << array[i] << " ";

    }

    std::cout << "\n";

}

template bool contains<int>(const int&, const int\*, size\_t);

template bool contains<double>(const double&, const double\*, size\_t);

template bool contains<std::string>(const std::string&, const std::string\*, size\_t);

template bool change<int>(const int&, const int&, int\*, size\_t);

template bool change<double>(const double&, const double&, double\*, size\_t);

template bool change<std::string>(const std::string&, const std::string&, std::string\*, size\_t);

template void printArray<int>(const int\*, size\_t);

template void printArray<double>(const double\*, size\_t);

template void printArray<std::string>(const std::string\*, size\_t);

array\_utils.h

#ifndef ARRAY\_UTILS\_H

#define ARRAY\_UTILS\_H

#include <iostream>

#include <string>

#include <utility> // Для std::swap

template <typename T>

bool contains(const T& element, const T\* array, size\_t size);

template <typename T>

bool change(const T& a, const T& b, T\* array, size\_t size);

template <typename T>

void printArray(const T\* array, size\_t size);

#endif // ARRAY\_UTILS\_H

Main.cpp

#include "array\_utils.h"

#include <iostream>

#include <string>

int main() {

    size\_t intSize = 5;

    int\* intArray = new int[intSize] {10, 20, 30, 40, 50};

    std::cout << "Array before swap: ";

    printArray(intArray, intSize);

    change(20, 40, intArray, intSize);

    std::cout << "Array after swap: ";

    printArray(intArray, intSize);

    change(20, 99, intArray, intSize);

    delete[] intArray;

    size\_t doubleSize = 4;

    double\* doubleArray = new double[doubleSize] {1.5, 2.7, 3.14, 5.0};

    std::cout << "\nArray before swap: ";

    printArray(doubleArray, doubleSize);

    change(2.7, 5.0, doubleArray, doubleSize);

    std::cout << "Array after swap: ";

    printArray(doubleArray, doubleSize);

    delete[] doubleArray;

    size\_t stringSize = 3;

    std::string\* stringArray = new std::string[stringSize] {"apple", "banana", "orange"};

    std::cout << "\nArray before swap: ";

    printArray(stringArray, stringSize);

    change(std::string("banana"), std::string("orange"), stringArray, stringSize);

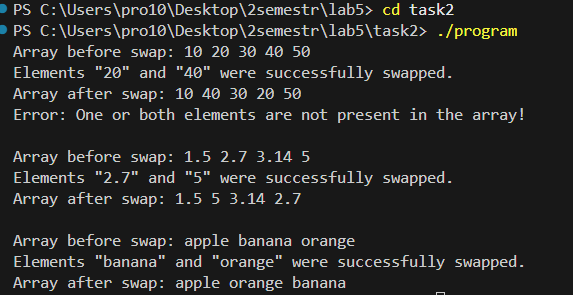
    std::cout << "Array after swap: ";

    printArray(stringArray, stringSize);

    delete[] stringArray;

    return 0;

}



## Задание 3.

Реализуйте шаблонный класс ***Matrix***. В строках матрицы должно быть одинаковое число элементов. Реализуйте конструкторы и деструктор класса, методы Set и Get для изменения и возврата значения элемента в i-ой строке и j-ом столбце матрицы. Определите операторы: “+” сложения двух матриц, “\*” умножения двух матриц, операции для потоковых ввода и вывода матриц “>>” и “<<”. Продемонстрируйте работу шаблонного класса и всех его методов на числовом и строковом типах данных.

Matrix.cpp

#include "matrix.h"

#include <type\_traits>

#include <stdexcept>

// Конструкторы

template <typename T>

Matrix<T>::Matrix(size\_t rows, size\_t cols) : rows(rows), cols(cols), data(rows, std::vector<T>(cols)) {}

template <typename T>

Matrix<T>::Matrix(size\_t rows, size\_t cols, const T& initValue) :

    rows(rows), cols(cols), data(rows, std::vector<T>(cols, initValue)) {}

// Получение размеров матрицы

template <typename T>

size\_t Matrix<T>::getRows() const { return rows; }

template <typename T>

size\_t Matrix<T>::getCols() const { return cols; }

// Доступ к элементам

template <typename T>

T& Matrix<T>::operator()(size\_t i, size\_t j) {

    if (i >= rows || j >= cols) {

        throw std::out\_of\_range("Matrix indices out of range");

    }

    return data[i][j];

}

template <typename T>

const T& Matrix<T>::operator()(size\_t i, size\_t j) const {

    if (i >= rows || j >= cols) {

        throw std::out\_of\_range("Matrix indices out of range");

    }

    return data[i][j];

}

// Методы Set и Get

template <typename T>

void Matrix<T>::Set(size\_t i, size\_t j, const T& value) {

    if (i >= rows || j >= cols) {

        throw std::out\_of\_range("Matrix indices out of range");

    }

    data[i][j] = value;

}

template <typename T>

T Matrix<T>::Get(size\_t i, size\_t j) const {

    if (i >= rows || j >= cols) {

        throw std::out\_of\_range("Matrix indices out of range");

    }

    return data[i][j];

}

// Оператор сложения матриц

template <typename T>

Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) const {

    if (rows != other.rows || cols != other.cols) {

        throw std::invalid\_argument("Matrix dimensions must agree for addition");

    }

    Matrix<T> result(rows, cols);

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

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

            result.data[i][j] = data[i][j] + other.data[i][j];

        }

    }

    return result;

}

// Оператор умножения матриц (специализация для строк)

template <>

Matrix<std::string> Matrix<std::string>::operator\*(const Matrix<std::string>& other) const {

    throw std::invalid\_argument("Matrix multiplication is not supported for strings");

}

// Оператор умножения матриц (общий случай)

template <typename T>

Matrix<T> Matrix<T>::operator\*(const Matrix<T>& other) const {

    if (cols != other.rows) {

        throw std::invalid\_argument("Matrix dimensions must agree for multiplication");

    }

    Matrix<T> result(rows, other.cols);

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

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

            T sum = T();

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

                sum += data[i][k] \* other.data[k][j];

            }

            result.data[i][j] = sum;

        }

    }

    return result;

}

// Операторы ввода/вывода

template <typename T>

std::ostream& operator<<(std::ostream& os, const Matrix<T>& matrix) {

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

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

            os << matrix.data[i][j] << " ";

        }

        os << "\n";

    }

    return os;

}

template <typename T>

std::istream& operator>>(std::istream& is, Matrix<T>& matrix) {

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

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

            is >> matrix.data[i][j];

        }

    }

    return is;

}

// Явная инстанциация шаблонов

template class Matrix<int>;

template class Matrix<double>;

template class Matrix<float>;

template class Matrix<std::string>;

matrix.h

#ifndef MATRIX\_H

#define MATRIX\_H

#include <vector>

#include <stdexcept>

#include <iostream>

template <typename T>

class Matrix {

private:

    size\_t rows;

    size\_t cols;

    std::vector<std::vector<T>> data;

public:

    // Конструкторы

    Matrix(size\_t rows, size\_t cols);

    Matrix(size\_t rows, size\_t cols, const T& initValue);

    // Деструктор

    ~Matrix() = default;

    // Получение размеров матрицы

    size\_t getRows() const;

    size\_t getCols() const;

    // Доступ к элементам

    T& operator()(size\_t i, size\_t j);

    const T& operator()(size\_t i, size\_t j) const;

    // Методы Set и Get

    void Set(size\_t i, size\_t j, const T& value);

    T Get(size\_t i, size\_t j) const;

    // Операторы

    Matrix operator+(const Matrix& other) const;

    Matrix operator\*(const Matrix& other) const;

    // Операторы ввода/вывода

    template <typename U>

    friend std::ostream& operator<<(std::ostream& os, const Matrix<U>& matrix);

    template <typename U>

    friend std::istream& operator>>(std::istream& is, Matrix<U>& matrix);

};

#include "matrix.cpp"

#endif // MATRIX\_H

Main.cpp

#include "matrix.h"

#include <iostream>

int main() {

    // Демонстрация работы с числовым типом (int)

    std::cout << "Integer Matrix Demonstration:\n";

    Matrix<int> intMat1(2, 2);

    intMat1.Set(0, 0, 1); intMat1.Set(0, 1, 2);

    intMat1.Set(1, 0, 3); intMat1.Set(1, 1, 4);

    Matrix<int> intMat2(2, 2);

    intMat2.Set(0, 0, 5); intMat2.Set(0, 1, 6);

    intMat2.Set(1, 0, 7); intMat2.Set(1, 1, 8);

    std::cout << "Matrix 1:\n" << intMat1;

    std::cout << "Matrix 2:\n" << intMat2;

    Matrix<int> intSum = intMat1 + intMat2;

    std::cout << "Sum:\n" << intSum;

    Matrix<int> intProd = intMat1 \* intMat2;

    std::cout << "Product:\n" << intProd;

    // Демонстрация работы со строковым типом

    std::cout << "\nString Matrix Demonstration:\n";

    Matrix<std::string> strMat1(2, 2);

    strMat1.Set(0, 0, "Hello"); strMat1.Set(0, 1, "World");

    strMat1.Set(1, 0, "C++");   strMat1.Set(1, 1, "Matrix");

    Matrix<std::string> strMat2(2, 2);

    strMat2.Set(0, 0, "Good");  strMat2.Set(0, 1, "Morning");

    strMat2.Set(1, 0, "STL");   strMat2.Set(1, 1, "Vector");

    std::cout << "Matrix 1:\n" << strMat1;

    std::cout << "Matrix 2:\n" << strMat2;

    Matrix<std::string> strSum = strMat1 + strMat2;

    std::cout << "Concatenation (sum):\n" << strSum;

    try {

        Matrix<std::string> strProd = strMat1 \* strMat2;

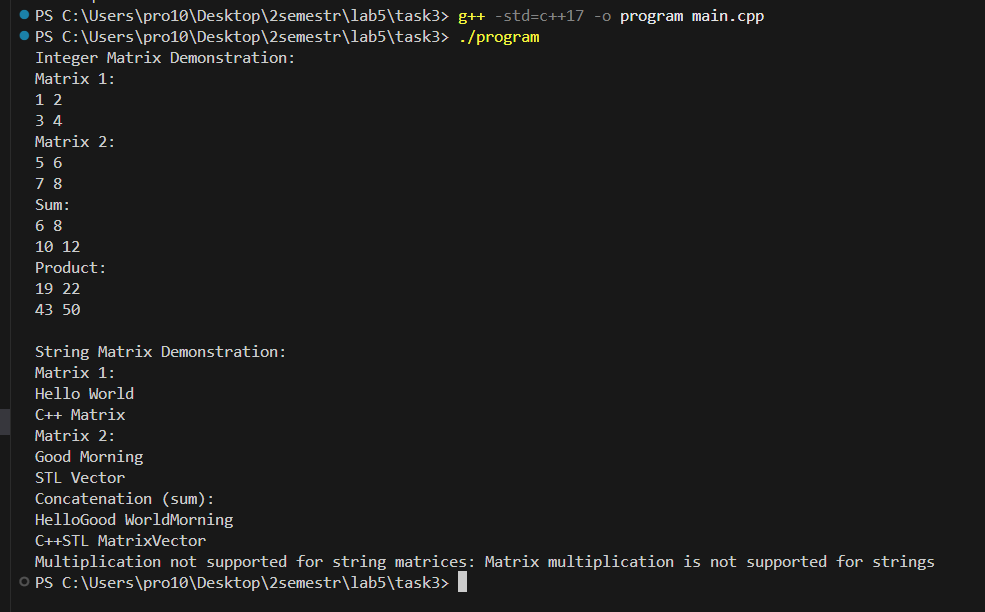
    } catch (const std::invalid\_argument& e) {

        std::cout << "Multiplication not supported for string matrices: " << e.what() << "\n";

    }

    return 0;

}



## Задание 4.

***Справка:***

*Многочлен от одной переменной – алгебраическое выражение, состоящие из суммы нескольких произведений числовых коэффициентов на переменную в натуральной степени.*

*Пример: x2 + 2x + 1. Слагаемыми в многочлене называют одночленами.*

*Так же, как и в задаче о матрице, числами здесь могут являться любые объекты со стандартным набором базовых математических операций (сложение, вычитание, умножение, деление), например, дробные, вещественные или комплексные числа, а так же математические матрицы и другие алгебраические объекты.*

Реализуйте шаблонный класс ***Polynomial*** (многочлен от одной переменной) на основе контейнера std::vector. Тип коэффициентов многочлена передавайте в качестве параметра шаблона. Хранение коэффициентов должно быть плотным (то есть должны храниться все коэффициенты, в том числе и промежуточные нулевые).

Сделайте следующее:

1. Напишите конструкторы, которые
   * создают многочлен по заданному вектору коэффициентов (коэффициенты задаются по возрастанию степени).
   * создают многочлен по заданному коэффициенту (многочлен нулевой степени), который равен значению по умолчанию параметра шаблона.
2. Перегрузите операторы == и !=. Ваш код должен быть очень простым. Операторы должны работать и в том случае, когда один из аргументов является скалярной величиной.

Complex.cpp

#include "Complex.h"

Complex::Complex() {

    re = new double(0);

    im = new double(0);

}

Complex::Complex(double x, double y) {

    this->re = new double(x);

    this->im = new double(y);

}

Complex::Complex(const Complex& z) {

    this->re = new double(\*z.re);

    this->im = new double(\*z.im);

}

Complex::~Complex() {

    delete re;

    delete im;

}

// Установка действительной части

void Complex::SetRe(double x) {

    \*re = x;

}

// Установка мнимой части

void Complex::SetIm(double y) {

    \*im = y;

}

// Получение действительной части

double Complex::GetRe() const {

    return \*re;

}

// Получение мнимой части

double Complex::GetIm() const {

    return \*im;

}

// Вычисление модуля комплексного числа

double Complex::Abs() const {

    return std::sqrt((\*re) \* (\*re) + (\*im) \* (\*im));    // Формула: sqrt(re^2 + im^2)

}

// Вычисление аргумента комплексного числа

double Complex::Arg() const {

    return std::atan2(\*im, \*re);    // Формула: atan2(im, re)

}

// Вывод числа в алгебраической форме

void Complex::Print() const {

    std::cout << \*re << " + i\*" << \*im << std::endl;    // Формат: a + bi

}

// Вывод числа в тригонометрической форме

void Complex::TrigPrint() const {

    double rho = Abs();

    double phi = Arg();

    std::cout << rho << " \* (cos(" << phi << ") + i\*sin(" << phi << "))" << std::endl;  // Формат: r \* (cos(phi) + i\*sin(phi))

}

// Вывод числа в экспоненциальной форме

void Complex::ExpPrint() const {

    double rho = Abs();

    double phi = Arg();

    std::cout << rho << " \* e^(i\*" << phi << ")" << std::endl;  // Формат: r \* e^(i\*phi)

}

Complex Complex::operator+(const Complex& z) const {

    return Complex(\*re + \*z.re, \*im + \*z.im);   // Формула: (a + bi) + (c + di) = (a + c) + (b + d)i

}

Complex Complex::operator-(const Complex& z) const {

    return Complex(\*re - \*z.re, \*im - \*z.im);   // Формула: (a + bi) + (c + di) = (a + c) + (b + d)i

}

Complex Complex::operator\*(const Complex& z) const {

    return Complex(\*re \* \*z.re - \*im \* \*z.im, \*re \* \*z.im + \*im \* \*z.re);   // Формула: (a + bi) \* (c + di) = (a\*c - b\*d) + (a\*d + b\*c)i   // Формула: (a + bi) + (c + di) = (a + c) + (b + d)i

}

Complex Complex::operator/(const Complex& z) const {

    double denominator = \*z.re \* \*z.re + \*z.im \* \*z.im; // Знаменатель: c^2 + d^2

    return Complex((\*re \* \*z.re + \*im \* \*z.im) / denominator, (\*im \* \*z.re - \*re \* \*z.im) / denominator);  // Формула: (a + bi) + (c + di) = (a + c) + (b + d)i

}

Complex &Complex::operator++() {

    \*this->re += 1;

    \*this->im += 1;

    return \*this;

}

Complex &Complex::operator--() {

    \*this->re -= 1;

    \*this->im -= 1;

    return \*this;

}

Complex &Complex::operator++(int) {

    \*this->re += 1;

    \*this->im += 1;

    return \*this;

}

Complex &Complex::operator--(int) {

    \*this->re -= 1;

    \*this->im -= 1;

    return \*this;

}

bool Complex::operator==(const Complex &z) const {

    return \*re == \*z.re && \*im == \*z.im;

}

bool Complex::operator!=(const Complex &z) const {

    return !(\*this == z);

}

bool Complex::operator<(const Complex &z) const {

    return this->Abs() < z.Abs();

}

bool Complex::operator>(const Complex &z) const {

    return this->Abs() > z.Abs();

}

bool Complex::operator<=(const Complex &z) const {

    return this->Abs() <= z.Abs();

}

bool Complex::operator>=(const Complex &z) const {

    return this->Abs() >= z.Abs();

}

// Сложение двух комплексных чисел

Complex Complex::Add(const Complex& z) const {

    return Complex(\*re + \*z.re, \*im + \*z.im);   // Формула: (a + bi) + (c + di) = (a + c) + (b + d)i

}

// Вычитание комплексных чисел

Complex Complex::Sub(const Complex& z) const {

    return Complex(\*re - \*z.re, \*im - \*z.im);   // Формула: (a + bi) - (c + di) = (a - c) + (b - d)i

}

// Умножение комплексных чисел

Complex Complex::Mult(const Complex& z) const {

    return Complex(\*re \* \*z.re - \*im \* \*z.im, \*re \* \*z.im + \*im \* \*z.re);   // Формула: (a + bi) \* (c + di) = (a\*c - b\*d) + (a\*d + b\*c)i

}

// Деление комплексных чисел

Complex Complex::Div(const Complex& z) const {

    double denominator = \*z.re \* \*z.re + \*z.im \* \*z.im; // Знаменатель: c^2 + d^2

    return Complex((\*re \* \*z.re + \*im \* \*z.im) / denominator, (\*im \* \*z.re - \*re \* \*z.im) / denominator);   // Формула: (a + bi) / (c + di) = [(a\*c + b\*d) + (b\*c - a\*d)i] / (c^2 + d^2)

}

Complex operator+(int number, const Complex& z) {

    return Complex(z.GetRe() + number, z.GetIm());

}

Complex operator+(const Complex& z, int number) {

    return Complex(z.GetRe() + number, z.GetIm());

}

Complex.h

#ifndef COMPLEX\_H

#define COMPLEX\_H

#include <iostream>

class Complex {

private:

    double real;

    double imaginary;

public:

    Complex(double r = 0.0, double i = 0.0) : real(r), imaginary(i) {}

    // Операторы сравнения

    bool operator==(const Complex& other) const {

        return real == other.real && imaginary == other.imaginary;

    }

    bool operator!=(const Complex& other) const {

        return !(\*this == other);

    }

    // Арифметические операторы

    Complex operator+(const Complex& other) const {

        return Complex(real + other.real, imaginary + other.imaginary);

    }

    Complex operator-(const Complex& other) const {

        return Complex(real - other.real, imaginary - other.imaginary);

    }

    Complex operator\*(const Complex& other) const {

        return Complex(

            real \* other.real - imaginary \* other.imaginary,

            real \* other.imaginary + imaginary \* other.real

        );

    }

    Complex& operator+=(const Complex& other) {

        real += other.real;

        imaginary += other.imaginary;

        return \*this;

    }

    Complex& operator-=(const Complex& other) {

        real -= other.real;

        imaginary -= other.imaginary;

        return \*this;

    }

    Complex& operator\*=(const Complex& other) {

        double new\_real = real \* other.real - imaginary \* other.imaginary;

        double new\_imaginary = real \* other.imaginary + imaginary \* other.real;

        real = new\_real;

        imaginary = new\_imaginary;

        return \*this;

    }

    Complex& operator\*=(double scalar) {

        real \*= scalar;

        imaginary \*= scalar;

        return \*this;

    }

    // Дружественная функция для вывода

    friend std::ostream& operator<<(std::ostream& os, const Complex& c) {

        os << "(" << c.real << ", " << c.imaginary << ")";

        return os;

    }

};

#endif // COMPLEX\_H

Main.cpp

#include "polynomial.h"

#include "complex.h"

#include <iostream>

int main() {

    // Тестирование с int

    Polynomial<int> p1({1, 2, 3}); // 3x^2 + 2x + 1

    std::cout << "p1: " << p1 << std::endl;

    std::cout << "Degree of p1: " << p1.Degree() << std::endl;

    std::cout << "p1(2): " << p1(2) << std::endl;

    // Тестирование с Complex

    Complex c1(1, 1), c2(2, -1), c3(0, 1);

    Polynomial<Complex> p2({c1, c2, c3}); // (0+1i)x^2 + (2-1i)x + (1+1i)

    std::cout << "\np2: " << p2 << std::endl;

    std::cout << "Degree of p2: " << p2.Degree() << std::endl;

    Complex x(1, 0);

    std::cout << "p2(1+0i): " << p2(x) << std::endl;

    // Операции с комплексными числами

    Polynomial<Complex> p3 = p2 + Complex(1, 0);

    std::cout << "\np2 + 1: " << p3 << std::endl;

    Polynomial<Complex> p4 = p2 \* Complex(0, 1);

    std::cout << "p2 \* i: " << p4 << std::endl;

    return 0;

}

Polynomial.h

#ifndef POLYNOMIAL\_H

#define POLYNOMIAL\_H

#include "complex.h"

#include <vector>

#include <iostream>

#include <algorithm>

#include <cmath>

template <typename T>

class Polynomial {

private:

    std::vector<T> coefficients; // Коэффициенты по возрастанию степеней

public:

    // Конструкторы

    Polynomial() : coefficients(1, T()) {} // Многочлен нулевой степени (значение по умолчанию)

    Polynomial(const T& scalar) : coefficients(1, scalar) {} // Многочлен нулевой степени с заданным значением

    Polynomial(const std::vector<T>& coeffs) : coefficients(coeffs) {} // По вектору коэффициентов

    // Операторы сравнения

    bool operator==(const Polynomial& other) const {

        return coefficients == other.coefficients;

    }

    bool operator!=(const Polynomial& other) const {

        return !(\*this == other);

    }

    template <typename U>

    bool operator==(const U& scalar) const {

        if (Degree() != 0) return false;

        return coefficients[0] == T(scalar);

    }

    template <typename U>

    bool operator!=(const U& scalar) const {

        return !(\*this == scalar);

    }

    // Арифметические операторы

    Polynomial operator+(const Polynomial& other) const {

        std::vector<T> result(std::max(coefficients.size(), other.coefficients.size()), T());

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

            if (i < coefficients.size()) result[i] += coefficients[i];

            if (i < other.coefficients.size()) result[i] += other.coefficients[i];

        }

        return Polynomial(result);

    }

    Polynomial operator-(const Polynomial& other) const {

        std::vector<T> result(std::max(coefficients.size(), other.coefficients.size()), T());

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

            if (i < coefficients.size()) result[i] += coefficients[i];

            if (i < other.coefficients.size()) result[i] -= other.coefficients[i];

        }

        return Polynomial(result);

    }

    Polynomial operator\*(const Polynomial& other) const {

        std::vector<T> result(coefficients.size() + other.coefficients.size() - 1, T());

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

            for (size\_t j = 0; j < other.coefficients.size(); ++j) {

                result[i + j] += coefficients[i] \* other.coefficients[j];

            }

        }

        return Polynomial(result);

    }

    Polynomial& operator+=(const Polynomial& other) {

        \*this = \*this + other;

        return \*this;

    }

    Polynomial& operator-=(const Polynomial& other) {

        \*this = \*this - other;

        return \*this;

    }

    Polynomial& operator\*=(const Polynomial& other) {

        \*this = \*this \* other;

        return \*this;

    }

    template <typename U>

    Polynomial operator+(const U& scalar) const {

        Polynomial result(\*this);

        if (result.coefficients.empty()) {

            result.coefficients.push\_back(T(scalar));

        } else {

            result.coefficients[0] += T(scalar);

        }

        return result;

    }

    template <typename U>

    Polynomial operator-(const U& scalar) const {

        Polynomial result(\*this);

        if (result.coefficients.empty()) {

            result.coefficients.push\_back(-T(scalar));

        } else {

            result.coefficients[0] -= T(scalar);

        }

        return result;

    }

    template <typename U>

    Polynomial operator\*(const U& scalar) const {

        Polynomial result(\*this);

        for (auto& coeff : result.coefficients) {

            coeff \*= T(scalar);

        }

        return result;

    }

    template <typename U>

    Polynomial& operator+=(const U& scalar) {

        if (coefficients.empty()) {

            coefficients.push\_back(T(scalar));

        } else {

            coefficients[0] += T(scalar);

        }

        return \*this;

    }

    template <typename U>

    Polynomial& operator-=(const U& scalar) {

        if (coefficients.empty()) {

            coefficients.push\_back(-T(scalar));

        } else {

            coefficients[0] -= T(scalar);

        }

        return \*this;

    }

    template <typename U>

    Polynomial& operator\*=(const U& scalar) {

        for (auto& coeff : coefficients) {

            coeff \*= T(scalar);

        }

        return \*this;

    }

    // Оператор доступа к коэффициенту

    const T& operator[](size\_t degree) const {

        static const T zero = T();

        if (degree >= coefficients.size()) return zero;

        return coefficients[degree];

    }

    // Метод для вычисления степени многочлена

    int Degree() const {

        if (coefficients.empty()) return -1; // Нулевой многочлен

        for (int i = coefficients.size() - 1; i >= 0; --i) {

            if (coefficients[i] != T()) return i;

        }

        return -1; // Все коэффициенты нулевые

    }

    // Оператор вычисления значения многочлена в точке

    T operator()(const T& x) const {

        T result = T();

        if (coefficients.empty()) return result;

        // Используем схему Горнера для эффективного вычисления

        for (int i = Degree(); i >= 0; --i) {

            result = result \* x + coefficients[i];

        }

        return result;

    }

    // Дружественная функция для вывода в поток

    friend std::ostream& operator<<(std::ostream& os, const Polynomial& poly) {

        int degree = poly.Degree();

        if (degree == -1) {

            os << "0";

            return os;

        }

        for (int i = degree; i >= 0; --i) {

            os << poly.coefficients[i];

            if (i != 0) os << " ";

        }

        return os;

    }

};

// Внешние операторы для скалярных операций

template <typename T, typename U>

Polynomial<T> operator+(const U& scalar, const Polynomial<T>& poly) {

    return poly + scalar;

}

template <typename T, typename U>

Polynomial<T> operator-(const U& scalar, const Polynomial<T>& poly) {

    return Polynomial<T>(scalar) - poly;

}

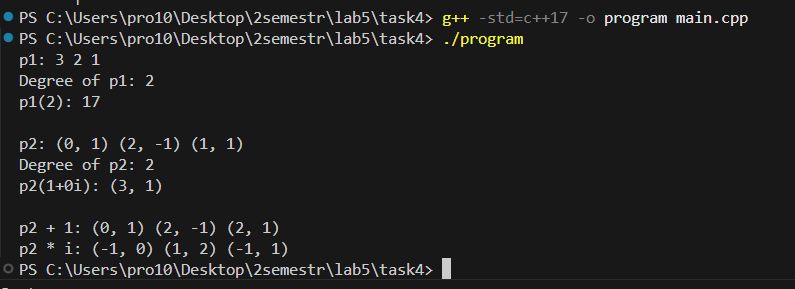
template <typename T, typename U>

Polynomial<T> operator\*(const U& scalar, const Polynomial<T>& poly) {

    return poly \* scalar;

}

#endif // POLYNOMIAL\_H



1. Перегрузите операторы +, - и \*, а также соответствующие операторы += и \*=. Учтите, что должны быть определены и такие арифметические операции, в которых один из аргументов является скалярной величиной.
2. Перегрузите оператор [] для получения коэффициента многочлена перед заданной степенью переменной. Достаточно константной версии этого оператора. Оператор должен работать для любых степеней (в том числе больше текущей максимальной).
3. Напишите метод Degree для вычисления степени многочлена.
4. Перегрузите оператор () для вычисления значения многочлена в точке. В качестве аргумента этот оператор принимает значение того типа, от которого создан многочлен. Постарайтесь написать эффективный код.
5. Перегрузите оператор << для печати многочлена в поток вывода. Для простоты будем выводить коэффициенты через пробел от **старшей степени к младшей**.
6. Протестируйте и продемонстрируйте работу класса для коэффициентов следующих типов: int, Complex (из лаб 3) или Rаtionаl(из лаб 3).

# Критерии оценки:

Если программа компилируется и оформлена по требованиям, то за лабораторную работу начисляются баллы.

Максимально можно получить 6 баллов:

* Задание 1 – 0,5 балла
* Задание 2 – 0,5 балла
* Задание 3 – 2 балла
* Задание 4 – 3 балла