TASK 1

#include <iostream>

#include <numeric> // Для std::gcd (наибольший общий делитель)

class Fraction {

private:

    int numerator;

    int denominator;

    // Вспомогательная функция для приведения к наименьшему общему знаменателю

    void simplify() {

        int common\_divisor = std::gcd(numerator, denominator);

        numerator /= common\_divisor;

        denominator /= common\_divisor;

        // Убедимся, что знаменатель всегда положительный

        if (denominator < 0) {

            numerator = -numerator;

            denominator = -denominator;

        }

    }

public:

    Fraction(int num = 0, int den = 1) : numerator(num), denominator(den) {

        if (denominator == 0) {

            std::cerr << "Ошибка: Знаменатель не может быть равен 0. Установлен знаменатель = 1." << std::endl;

            denominator = 1;

        }

        simplify();

    }

    // Оператор сложения

    Fraction operator+(const Fraction& other) const {

        int new\_numerator = numerator \* other.denominator + other.numerator \* denominator;

        int new\_denominator = denominator \* other.denominator;

        return Fraction(new\_numerator, new\_denominator);

    }

    // Оператор вычитания

    Fraction operator-(const Fraction& other) const {

        int new\_numerator = numerator \* other.denominator - other.numerator \* denominator;

        int new\_denominator = denominator \* other.denominator;

        return Fraction(new\_numerator, new\_denominator);

    }

    // Оператор умножения

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

        int new\_numerator = numerator \* other.numerator;

        int new\_denominator = denominator \* other.denominator;

        return Fraction(new\_numerator, new\_denominator);

    }

    // Оператор деления

    Fraction operator/(const Fraction& other) const {

        if (other.numerator == 0) {

            std::cerr << "Ошибка: Деление на ноль." << std::endl;

            return Fraction(0, 1); // Возвращаем 0/1 в случае ошибки

        }

        int new\_numerator = numerator \* other.denominator;

        int new\_denominator = denominator \* other.numerator;

        return Fraction(new\_numerator, new\_denominator);

    }

    // Дружественная функция для вывода дроби в поток

    friend std::ostream& operator<<(std::ostream& os, const Fraction& fraction) {

        os << fraction.numerator << "/" << fraction.denominator;

        return os;

    }

};

int main() {

    Fraction f1(1, 2);

    Fraction f2(1, 4);

    std::cout << "f1 = " << f1 << std::endl;

    std::cout << "f2 = " << f2 << std::endl;

    std::cout << "f1 + f2 = " << f1 + f2 << std::endl;

    std::cout << "f1 - f2 = " << f1 - f2 << std::endl;

    std::cout << "f1 \* f2 = " << f1 \* f2 << std::endl;

    std::cout << "f1 / f2 = " << f1 / f2 << std::endl;

    Fraction f3(5, 10);  // Пример с необходимостью упрощения

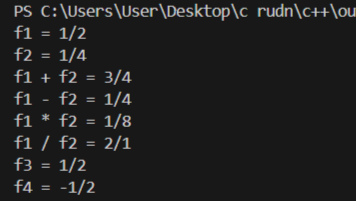
    std::cout << "f3 = " << f3 << std::endl; // Вывод: 1/2

    Fraction f4(1, -2); // Пример с отрицательным знаменателем

    std::cout << "f4 = " << f4 << std::endl; // Вывод: -1/2

    return 0;

}



TASK 2

#include <iostream>

#include <string>

class Student {

    public:

        std::string name;

        int grade;

        Student(const std::string& name, int grade) : name(name), grade(grade) {}

         // Перегрузка оператора ==

         // Cравнимаем студентов по оценке

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

        return this->grade == other.grade;

    }

    // Перегрузка оператора <

    // Сравниваем студентов по алфавиту (по имени)

    bool operator<(const Student& other) const {

        return this->name < other.name;

    }

};

int main() {

    Student s1("Ivan", 90);

    Student s2("Anna", 90);

    Student s3("Petr", 85);

    std::cout << std::boolalpha;

    // Сравнение по оценке

    std::cout << "s1 == s2: " << (s1 == s2) << std::endl; // true, т.к. оценки равны

    std::cout << "s1 == s3: " << (s1 == s3) << std::endl; // false

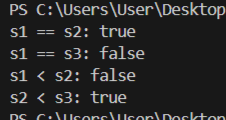
    // Сравнение по имени

    std::cout << "s1 < s2: " << (s1 < s2) << std::endl;   // false, т.к. "Ivan" > "Anna"

    std::cout << "s2 < s3: " << (s2 < s3) << std::endl;   // true, т.к. "Anna" < "Petr"

    return 0;

}



TASK 3

#include <iostream>

#include <string>

class Book {

private:

    std::string title;

    int year;

public:

    Book(const std::string& t, int y) : title(t), year(y) {}

//перегрузка оператора вывода

friend std::ostream& operator<<(std::ostream& os, const Book& book){

    os<<"Title: "<< book.title << ", Year: " << book.year;

    return os;

}

//перегрузка оператора ввода

friend std::istream& operator>>(std::istream& in, Book& book){

    std::cout << "Введите заголовок: ";

    std::getline(in >>std::ws, book.title);

    std::cout<< "Enter year: ";

    in >> book.year;

    return in;

    }

};

int main(){

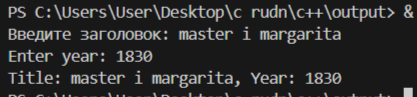
    Book book("", 0);

    std::cin >> book;

    std::cout << book << std::endl;

    return 0;

}



TASK 4

#include <iostream>

class Counter {

private:

    int count;  // текущее значение счетчика

public:

    Counter(int initial\_value = 0) : count(initial\_value) {}

    // Префиксный инкремент++

    Counter& operator++() {

        ++count;

        return \*this;

    }

    // Постфиксный инкремент

    Counter operator++(int) {

        Counter temp = \*this;

        ++count;

        return temp;

    }

    // Метод для получения текущего значения

    int getCount() const {

        return count;

    }

};

int main() {

    Counter c1(5);

    // Префиксный инкремент

    std::cout << "Исходное значение: " << c1.getCount() << std::endl;

    Counter c2 = ++c1;

    std::cout << "После префиксного ++: " << c1.getCount() << std::endl;

    std::cout << "Результат префиксного ++: " << c2.getCount() << std::endl;

    // Постфиксный инкремент

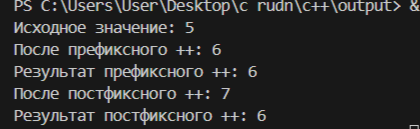
    Counter c3 = c1++;

    std::cout << "После постфиксного ++: " << c1.getCount() << std::endl;

    std::cout << "Результат постфиксного ++: " << c3.getCount() << std::endl;

    return 0;

}



TASK 5

#include <iostream>

class Distance {

private:

    double meters;

public:

    Distance(double m = 0) : meters(m) {}

    // Оператор приведения к double (в километрах)

    operator double() const {

        return meters / 1000.0;

    }

};

int main() {

    Distance d1(2500);

    // Неявное преобразование в километры

    double kilometers = d1;

    std::cout << "2500 meters = " << kilometers << " km\n";

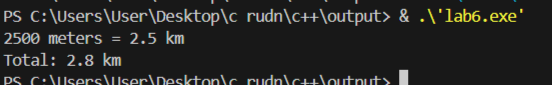
    Distance d2(300);

    double total\_km = d1 + d2;  // 2500m + 300m = 2.8 km

    std::cout << "Total: " << total\_km << " km\n";

    return 0;

}



TASK 6

#include <iostream>

class Rectangle {

private:

    double width;

    double height;

public:

    Rectangle(double w = 0, double h = 0) : width(w), height(h) {}

    friend double calculateArea(const Rectangle& rect);

};

double calculateArea(const Rectangle& rect) {

    return rect.width \* rect.height;

}

int main(){

    Rectangle rect(4.0, 2.0);

    std::cout << "Площадь: " << calculateArea(rect) << std::endl;

    return 0;

}



TASK 7

#include <iostream>

#include <string>

class SecretAccess;

class Secret {

private:

    std::string data;

public:

    Secret(const std::string& d) : data(d) {}

    friend class SecretAccess;

};

class SecretAccess {

public:

    void printSecret(const Secret& secret){

        std::cout << "Secret data: " << secret.data << std::endl;

    }

};

int main() {

    Secret secret("password");

    SecretAccess access;

    access.printSecret(secret);

    return 0;

}



TASK 8

#include <iostream>

#include <algorithm> // для std::copy

class DynamicArray {

private:

    int\* data;      // Указатель на динамический массив

    size\_t size;    // Размер массива

public:

    DynamicArray(size\_t size = 0) : size(size) {

        data = (size > 0) ? new int[size] : nullptr;

    }

    ~DynamicArray() {

        delete[] data;

    }

    // Конструктор копирования (нужен для правильной работы оператора присваивания)

    DynamicArray(const DynamicArray& other) : size(other.size) {

        data = (size > 0) ? new int[size] : nullptr;

        std::copy(other.data, other.data + size, data);

    }

    // Перегрузка оператора присваивания

    DynamicArray& operator=(const DynamicArray& other) {

        if (this != &other) { // Проверка на самоприсваивание

            // 1. Удаляем старые данные

            delete[] data;

            // 2. Копируем размер

            size = other.size;

            // 3. Выделяем новую память и копируем данные

            data = (size > 0) ? new int[size] : nullptr;

            std::copy(other.data, other.data + size, data);

        }

        return \*this;

    }

    // Метод для доступа к элементам

    int& operator[](size\_t index) {

        return data[index];

    }

    // Метод для получения размера

    size\_t getSize() const {

        return size;

    }

    // Метод для вывода массива

    void print() const {

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

            std::cout << data[i] << " ";

        }

        std::cout << std::endl;

    }

};

int main() {

    DynamicArray arr1(5);

    for (size\_t i = 0; i < arr1.getSize(); ++i) {

        arr1[i] = static\_cast<int>(i) \* 10;

    }

    DynamicArray arr2;

    arr2 = arr1; // Используем перегруженный оператор присваивания

    std::cout << "Array 1: ";

    arr1.print();

    std::cout << "Array 2 (copy): ";

    arr2.print();

    // Меняем arr1 и проверяем, что arr2 не изменился

    arr1[0] = 100;

    std::cout << "\nAfter modification:\n";

    std::cout << "Array 1: ";

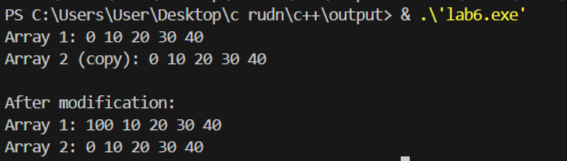
    arr1.print();

    std::cout << "Array 2: ";

    arr2.print();

    return 0;

}



TASK 9

#include <iostream>

#include <cstring>

class String {

private:

    char\* data; //указатель на массив символов

    int length;

public:

    String(const char\* s = "") {

        length = strlen(s);

        data = new char[length + 1];

        strcpy(data, s);

    }

    ~String() {

        delete[] data;

    }

    // Оператор [] для чтения и записи

    char& operator[](int index) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Неверный индекс");

        }

        return data[index];

    }

    // Оператор [] только для чтения (константная версия)

    const char& operator[](int index) const {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Неверный индекс");

        }

        return data[index];

    }

    // Получение длины строки

    int getLength() const {

        return length;

    }

    // Вывод строки

    void print() const {

        std::cout << data << std::endl;

    }

};

int main() {

    // Создаем строку

    String s("Hello");

    // Пример чтения символов

    std::cout << "Первый символ: " << s[0] << std::endl;

    std::cout << "Второй символ: " << s[1] << std::endl;

    // Пример изменения символов

    s[0] = 'J';

    s[1] = 'a';

    std::cout << "После изменений: ";

    s.print();

    // Пример с константной строкой

    const String cs("World");

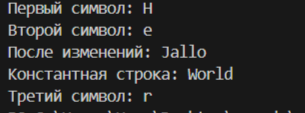
    std::cout << "Константная строка: ";

    cs.print();

    std::cout << "Третий символ: " << cs[2] << std::endl;

    return 0;

}



TASK 10

//task 10

#include <iostream>

#include <stdexcept>

class Matrix {

private:

    double data[2][2]; // Хранение элементов матрицы

public:

    // Конструктор по умолчанию (нулевая матрица)

    Matrix() {

        for (int i = 0; i < 2; ++i) {

            for (int j = 0; j < 2; ++j) {

                data[i][j] = 0.0;

            }

        }

    }

    // Конструктор с инициализацией

    Matrix(double a, double b, double c, double d) {

        data[0][0] = a;

        data[0][1] = b;

        data[1][0] = c;

        data[1][1] = d;

    }

    // Оператор сложения матриц

    Matrix operator+(const Matrix& other) const {

        Matrix result;

        for (int i = 0; i < 2; ++i) {

            for (int j = 0; j < 2; ++j) {

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

            }

        }

        return result;

    }

    // Оператор умножения матриц

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

        Matrix result;

        for (int i = 0; i < 2; ++i) {

            for (int j = 0; j < 2; ++j) {

                for (int k = 0; k < 2; ++k) {

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

                }

            }

        }

        return result;

    }

    // Оператор доступа к элементам матрицы

    double& operator()(int row, int col) {

        if (row < 0 || row >= 2 || col < 0 || col >= 2) {

            throw std::out\_of\_range("Неверные индексы матрицы");

        }

        return data[row][col];

    }

    // Константная версия оператора доступа

    const double& operator()(int row, int col) const {

        if (row < 0 || row >= 2 || col < 0 || col >= 2) {

            throw std::out\_of\_range("Неверные индексы матрицы");

        }

        return data[row][col];

    }

    // Оператор вывода матрицы

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

        os << "[(" << matrix.data[0][0] << ", " << matrix.data[0][1] << "),\n";

        os << " (" << matrix.data[1][0] << ", " << matrix.data[1][1] << ")]";

        return os;

    }

};

int main() {

    // Создаем матрицы

    Matrix m1(1, 2, 3, 4);

    Matrix m2(5, 6, 7, 8);

    std::cout << "Матрица 1:\n" << m1 << "\n\n";

    std::cout << "Матрица 2:\n" << m2 << "\n\n";

    Matrix sum = m1 + m2;

    std::cout << "Сумма матриц:\n" << sum << "\n\n";

    Matrix product = m1 \* m2;

    std::cout << "Произведение матриц:\n" << product << "\n\n";

    // Доступ к элементам

    std::cout << "Элемент (0,1) матрицы 1: " << m1(0, 1) << "\n";

    m1(1, 0) = 9; // Изменение элемента

    std::cout << "Измененная матрица 1:\n" << m1 << "\n";

    return 0;

}

