TASK 1

//task 1

#include <iostream>

#include <string>

class Vehice{

public:

    void start(){

        std::cout<<"Vehicle started"<<std::endl;

    }

    void stop(){

        std::cout<<"Vehicle stopped"<<std::endl;

    }

};

class Car: public Vehice{

public:

    void honk(){

        std::cout<<"BEEP!"<<std::endl;

    }

};

class Bicycle: public Vehice {

public:

    void ringBell() {

        std::cout<<"DING DING!"<<std::endl;

    }

};

int main(){

    Car myCar;

    Bicycle myBicycle;

    myCar.start();

    myCar.honk();

    myCar.stop();

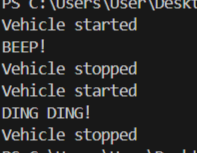
    myBicycle.start();

    myBicycle.ringBell();

    myBicycle.stop();

    return 0;

}



TASK 2

#include <iostream>

#include <vector>

// Базовый абстрактный класс Shape

class Shape {

    public:

        virtual void draw() const = 0;

};

class Circle : public Shape {

    public:

    void draw() const override {

        std::cout<<"Drawing a Circle"<<std::endl;

    }

};

class Square : public Shape {

    public:

        void draw() const override {

            std::cout << "Drawing a Square"<<std::endl;

        }

};

class Triangle : public Shape {

    public:

        void draw() const override {

            std::cout << "Drawing a Triangle"<<std::endl;;

        }

};

int main() {

    // Создаем объекты разных фигур

    Circle circle;

    Square square;

    Triangle triangle;

    std::vector<Shape\*> shapes = {&circle, &square, &triangle};

    std::cout << "Drawing all shapes:\n";

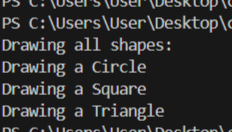
    for (Shape\* shape : shapes) {

        shape->draw();  // Вызывается правильная версия метода

    }

    return 0;

}



TASK 3

#include <iostream>

#include <vector>

#include <memory>

class Animal {

public:

    virtual void makeSound() const = 0;

};

class Dog : public Animal {

public:

    void makeSound() const override {

        std::cout << "Woof! Woof!"<<std::endl;

    }

};

class Cat : public Animal {

public:

    void makeSound() const override {

        std::cout << "Meow! Meow!"<<std::endl;

    }

};

class Cow : public Animal {

public:

    void makeSound() const override {

        std::cout << "Moo! Moo!"<<std::endl;

    }

};

void animalConcert(const std::vector<Animal\*>& animals) {

    for (const auto\* animal : animals) {

        animal->makeSound();

    }

}

int main() {

    Dog dog;

    Cat cat;

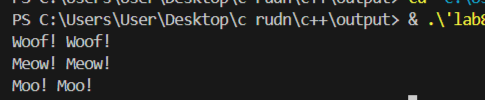
    Cow cow;

    std::vector<Animal\*> animals = {&dog, &cat, &cow};

    animalConcert(animals);

    return 0;

}



TASK 4

#include <iostream>

#include <string>

class Person {

protected:

    std::string name;

    int age;

public:

    Person(std::string n, int a) : name(n), age(a) {}

    void show() {

        std::cout << name << ", " << age << " лет"<<std::endl;;

    }

};

class Employee : public Person {

protected:

    int id;

    double salary;

public:

    Employee(std::string n, int a, int i, double s)

        : Person(n, a), id(i), salary(s) {}

    void show() {

        Person::show();

        std::cout << "ID: " << id << ", Зарплата: $" << salary << std::endl;

    }

    void work() {

        std::cout << name << " работает" << std::endl;

    }

};

class Manager : public Employee {

private:

    std::string department;

public:

    Manager(std::string n, int a, int i, double s, std::string d)

        : Employee(n, a, i, s), department(d) {}

    void show() {

        Employee::show();

        std::cout << "Отдел: " << department << "\n";

    }

    void manage() {

        std::cout << name << " управляет отделом " << department << "\n";

    }

};

int main() {

    Employee emp("Иван", 30, 101, 50000);

    Manager boss("Анна", 35, 201, 80000, "Продажи");

    emp.show();

    emp.work();

    std::cout << "\n";

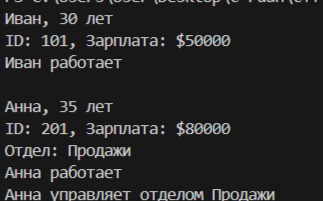
    boss.show();

    boss.work();

    boss.manage();

    return 0;

}



TASK 5

#include <iostream>

class Printer {

    public:

    void print(){

        std::cout << "Печатаем документ" << std::endl;

    }

};

class Scanner {

    public:

    void scan(){

        std::cout << "Сканируем документ" << std::endl;

    }

};

class Copier: public Printer, public Scanner {

    public:

    void copy(){

        scan();

        print();

        std::cout << "Документ скопирован!" << std::endl;

    }

};

int main() {

    Copier officeCopier;

    std::cout << "Используем как принтер" << std::endl;

    officeCopier.print();

    std::cout << "Используем как сканер" << std::endl;

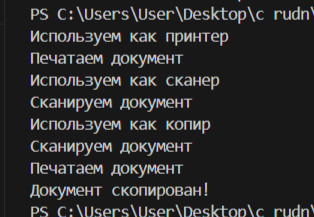
    officeCopier.scan();

    std::cout << "Используем как копир" << std::endl;

    officeCopier.copy();

    return 0;

}



TASK 6

#include <iostream>

class Base {

public:

    Base() { std::cout << "Base конструктор\n"; }

    ~Base() { std::cout << "Base деструктор\n"; }  // Не виртуальный

    //virtual ~Base() { std::cout << "Base деструктор\n"; }

};

class Derived : public Base {

private:

    int\* data;

public:

    Derived() : Base() {

        data = new int[100];

        std::cout << "Derived конструктор\n";

    }

    ~Derived() {

        delete[] data;  //при невиртулаьном деструкторе не вызовется

        std::cout << "Derived деструктор\n";

    }

};

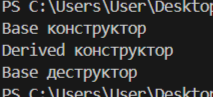
int main() {

    Base\* obj = new Derived();  // Создаем Derived через указатель на Base

    delete obj;

    return 0;

}



TASK 7

#include <iostream>

class Drawable {

    public:

        virtual void draw() const = 0;

        virtual double getArea() const = 0;

};

class Rectangle : public Drawable {

    double width, height;

public:

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

    void draw() const override {

        std::cout << "Рисуем прямоугольник "<< width << "x" << height << std::endl;

    }

    double getArea() const override {

        return width \* height;

    }

};

class Circle : public Drawable {

    double radius;

public:

    Circle(double r) : radius(r) {}

    void draw() const override {

        std::cout << "Рисуем круг радиусом "

                  << radius << std::endl;

    }

    double getArea() const override {

        return 3.14159 \* radius \* radius;

    }

};

void printShape(const Drawable& shape) {

    shape.draw();

    std::cout << "Площадь: " << shape.getArea() << std::endl;

}

int main() {

    Rectangle rect(5, 3);

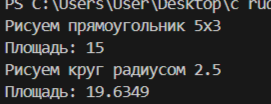
    Circle circle(2.5);

    printShape(rect);

    printShape(circle);

    return 0;

}



TASK 8

//task 8

#include <iostream>

class Base {

public:

    int publicVar = 1;

protected:

    int protectedVar = 2;

private:

    int privateVar = 3;

};

class DerivedPublic : public Base {

public:

    void show() {

        std::cout << publicVar << std::endl;

        std::cout << protectedVar << std::endl;;

    }

};

class DerivedProtected : protected Base {

public:

    void show() {

        std::cout << publicVar << std::endl;

        std::cout << protectedVar << std::endl;

    }

};

class DerivedPrivate : private Base {

public:

    void show() {

        std::cout << publicVar << std::endl;

        std::cout << protectedVar std::endl;

    }

};

int main() {

    DerivedPublic pub;

    pub.publicVar = 10;

    DerivedProtected prot;

    //prot.publicVar = 20;  // Ошибка: теперь protected

    DerivedPrivate priv;

    // priv.publicVar = 30;  // Ошибка: теперь private

    return 0;

}

TASK 9

#include <iostream>

// Базовый класс

class Engine {

public:

    void start() const {

        std::cout << "Двигатель запущен"<<std::endl;

    }

};

class Car : public Engine {

    public:

        void drive() const {

            std::cout << "Машина поехала"<<std::endl;

        }

    };

void demoInheritance() {

    std::cout << "Наследование"<<std::endl;;

    Car car;

    car.start();  // Машина "является" двигателем? Нет!

    car.drive();

}

class CarC {

    private:

        Engine engine;  // Композиция!

    public:

        void start() const {

            engine.start();  // Делегируем вызов двигателю

        }

        void drive() const {

            std::cout << "Машина поехала"<<std::endl;

        }

    };

void demoComposition() {

    std::cout << "Композиция"<<std::endl;

    CarC car;

    car.start();  // Машина "имеет" двигатель

    car.drive();

}

//Композиция предпочтительнее наследования, если нет чёткого отношения "является"

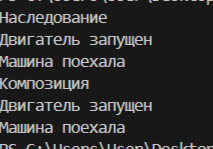
int main() {

    demoInheritance();

    demoComposition();

    return 0;

}



TASK 10

#include <iostream>

class Base {

public:

    virtual void foo() {

        std::cout << "Base::foo()"<<std::endl;

    }

    virtual void bar() final {

        std::cout << "Base::bar()"<<std::endl;

    }

};

class Derived : public Base {

public:

    void foo() override {

        std::cout << "Derived::foo()"<<std::endl;

    }

    // Нельзя переопределить bar(), так как он final

    // void bar() override {

        // std::cout << "Derived::bar()"<< std::endl;

    // }

};

// class Derived1 : public Derived {

    // public:

    //     void foo() override {

    //         std::cout << "Derived1::foo()"<<std::endl;

    //     }

    // };

int main() {

    Derived derived;

    derived.foo();

    derived.bar();  //не переопределен

    return 0;

}

