```cpp
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <fstream>
#include <memory> // For smart pointers

// 1. Class and Object Basics (Easy)
// Problem: Create a class Student with attributes name and roll_no. Take input from the user and display the student's details.
class Student {
public:
    std::string name;
    int roll_no;

    void displayDetails() {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Roll No: " << roll_no << std::endl;
    }
};

void solveClassAndObjectBasics() {
    std::cout << "\n--- 1. Class and Object Basics ---" << std::endl;
    Student s;
    std::cout << "Enter student name: ";
    std::cin >> s.name;
    std::cout << "Enter roll number: ";
    std::cin >> s.roll_no;
    s.displayDetails();
}
```

```
// 2. Constructor Initialization (Easy)
// Problem: Write a class Book with a parameterized constructor to initialize title and price. Print the book details.
class Book {
public:
    std::string title;
    double price;

    Book(std::string t, double p) : title(t), price(p) {}

    void displayDetails() {
        std::cout << "Title: " << title << std::endl;
        std::cout << "Price: " << price << std::endl;
    }
};

void solveConstructorInitialization() {
    std::cout << "\n--- 2. Constructor Initialization ---" << std::endl;
    Book b("The C++ Programming Language", 45.99);
    b.displayDetails();
}

// 3. Single Inheritance (Easy)
// Problem: Create two classes: Person (with name) and Employee (inherits from Person, with salary). Display both attributes.
class Person {
public:
    std::string name;

    Person(std::string n) : name(n) {}
```

```
    void displayPerson() {
        std::cout << "Name: " << name << std::endl;
    }
};

class Employee : public Person {
public:
    double salary;

    Employee(std::string n, double s) : Person(n), salary(s) {}

    void displayEmployee() {
        displayPerson();
        std::cout << "Salary: " << salary << std::endl;
    }
};

void solveSingleInheritance() {
    std::cout << "\n--- 3. Single Inheritance ---" << std::endl;
    Employee e("Alice", 60000.0);
    e.displayEmployee();
}

// 4. Multilevel Inheritance (Medium)
// Problem: Create a base class Vehicle, a derived class Car, and a further derived class ElectricCar. Add a method to print the complete info.
class Vehicle {
public:
    std::string model;

    Vehicle(std::string m) : model(m) {}
```

```
    void displayVehicleInfo() {
        std::cout << "Model: " << model;
    }
};

class Car : public Vehicle {
public:
    int numDoors;

    Car(std::string m, int doors) : Vehicle(m), numDoors(doors) {}

    void displayCarInfo() {
        displayVehicleInfo();
        std::cout << ", Number of Doors: " << numDoors;
    }
};

class ElectricCar : public Car {
public:
    double batteryCapacity;

    ElectricCar(std::string m, int doors, double capacity) : Car(m, doors),
batteryCapacity(capacity) {}

    void displayCompleteInfo() {
        displayCarInfo();
        std::cout << ", Battery Capacity: " << batteryCapacity << " kWh" << std::endl;
    }
};
```

```
void solveMultilevelInheritance() {
    std::cout << "\n--- 4. Multilevel Inheritance ---" << std::endl;
    ElectricCar ec("Tesla Model 3", 4, 75.0);
    ec.displayCompleteInfo();
}

// 5. Method Overloading (Easy)
// Problem: Create a class Calculator with overloaded methods add() for adding two integers, three integers, and two floats.
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }

    float add(float a, float b) {
        return a + b;
    }
};

void solveMethodOverloading() {
    std::cout << "\n--- 5. Method Overloading ---" << std::endl;
    Calculator calc;
    std::cout << "Adding 2 integers: " << calc.add(5, 3) << std::endl;
    std::cout << "Adding 3 integers: " << calc.add(5, 3, 2) << std::endl;
    std::cout << "Adding 2 floats: " << calc.add(2.5f, 3.7f) << std::endl;
}
```

```
// 6. Method Overriding & Polymorphism (Medium)
// Problem: Create a base class Shape with method draw(). Override this method in derived classes Circle and Rectangle. Use a pointer to call draw() based on object type.
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

void solveMethodOverridingPolymorphism() {
    std::cout << "\n--- 6. Method Overriding & Polymorphism ---" << std::endl;
    Shape* s1 = new Circle();
    Shape* s2 = new Rectangle();

    s1->draw(); // Calls Circle::draw()
```

```
    s2->draw(); // Calls Rectangle::draw()

    delete s1;
    delete s2;
}

// 7. Operator Overloading (Medium)
// Problem: Overload the + operator to add two complex numbers using a Complex class.
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

void solveOperatorOverloading() {
    std::cout << "\n--- 7. Operator Overloading ---" << std::endl;
    Complex c1(2.5, 3.0);
    Complex c2(1.5, 2.0);
    Complex sum = c1 + c2;
    std::cout << "Complex number 1: ";
```

```
    c1.display();
    std::cout << "Complex number 2: ";
    c2.display();
    std::cout << "Sum: ";
    sum.display();
}

// 8. Abstract Class & Interface-like Behavior (Medium)
// Problem: Create an abstract class Animal with a pure virtual function makeSound(). Derive Dog and Cat from it and override the function.
class Animal {
public:
    virtual void makeSound() = 0; // Pure virtual function
    virtual ~Animal() {} // Virtual destructor for proper cleanup
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Woof!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Meow!" << std::endl;
    }
};

void solveAbstractClass() {
```

```
    std::cout << "\n--- 8. Abstract Class & Interface-like Behavior ---" << std::endl;
    // Animal* animal; // Cannot create an object of an abstract class
    Dog d;
    Cat c;

    d.makeSound();
    c.makeSound();
}

// 9. Constructor Overloading (Easy)
// Problem: Write a class Box with overloaded constructors: one with no parameters, one with width and height, and one with all dimensions.
class Box {
public:
    double width;
    double height;
    double depth;

    Box() : width(0), height(0), depth(0) {
        std::cout << "Default constructor called." << std::endl;
    }

    Box(double w, double h) : width(w), height(h), depth(0) {
        std::cout << "Constructor with width and height called." << std::endl;
    }

    Box(double w, double h, double d) : width(w), height(h), depth(d) {
        std::cout << "Constructor with all dimensions called." << std::endl;
    }

    void displayDimensions() const {
```

```
        std::cout << "Width: " << width << ", Height: " << height << ", Depth: " << depth << std::endl;
    }
};

void solveConstructorOverloading() {
    std::cout << "\n--- 9. Constructor Overloading ---" << std::endl;
    Box b1;
    Box b2(5.0, 3.0);
    Box b3(2.0, 4.0, 6.0);

    b1.displayDimensions();
    b2.displayDimensions();
    b3.displayDimensions();
}

// 10. Exception Handling (Medium)
// Problem: Take two integers from the user and perform division. Handle the case where the denominator is zero using try-catch.
void solveExceptionHandling() {
    std::cout << "\n--- 10. Exception Handling ---" << std::endl;
    int numerator, denominator;
    std::cout << "Enter numerator: ";
    std::cin >> numerator;
    std::cout << "Enter denominator: ";
    std::cin >> denominator;

    try {
        if (denominator == 0) {
            throw std::runtime_error("Division by zero!");
        }
        double result = static_cast<double>(numerator) / denominator;
```

```
        std::cout << "Result of division: " << result << std::endl;
    } catch (const std::runtime_error& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }
}

// 11. Encapsulation Using Getters and Setters (Easy)
// Problem: Create a class BankAccount with private data members accountNumber and balance. Provide getter and setter methods to access and update them.
class BankAccount {
private:
    std::string accountNumber;
    double balance;

public:
    BankAccount(std::string accNum, double bal) : accountNumber(accNum), balance(bal) {}

    std::string getAccountNumber() const {
        return accountNumber;
    }

    double getBalance() const {
        return balance;
    }

    void setBalance(double newBalance) {
        balance = newBalance;
    }
};

void solveEncapsulation() {
```

```
    std::cout << "\n--- 11. Encapsulation Using Getters and Setters ---" << std::endl;
    BankAccount account("123456789", 1000.0);
    std::cout << "Account Number: " << account.getAccountNumber() << std::endl;
    std::cout << "Initial Balance: " << account.getBalance() << std::endl;
    account.setBalance(1500.0);
    std::cout << "Updated Balance: " << account.getBalance() << std::endl;
}

// 12. Protected Access Modifier (Easy)
// Problem: Create a base class Parent with a protected member familyName. Inherit a class Child and display the family name using it.
class Parent {
protected:
    std::string familyName;

public:
    Parent(std::string name) : familyName(name) {}
};

class Child : public Parent {
public:
    Child(std::string name) : Parent(name) {}

    void displayFamilyName() {
        std::cout << "Family Name: " << familyName << std::endl;
    }
};

void solveProtectedAccessModifier() {
    std::cout << "\n--- 12. Protected Access Modifier ---" << std::endl;
    Child c("Smith");
```

```
    c.displayFamilyName();
}

// 13. Hierarchical Inheritance (Medium)
// Problem: Create a base class Employee, and two derived classes Manager and Clerk. Display all relevant information using hierarchical inheritance.
class EmployeeBase {
public:
    std::string name;
    int employeeId;

    EmployeeBase(std::string n, int id) : name(n), employeeId(id) {}

    void displayBaseInfo() const {
        std::cout << "Name: " << name << ", ID: " << employeeId;
    }
};

class Manager : public EmployeeBase {
public:
    std::string department;

    Manager(std::string n, int id, std::string dept) : EmployeeBase(n, id), department(dept) {}

    void displayManagerInfo() const {
        displayBaseInfo();
        std::cout << ", Department: " << department << std::endl;
    }
};

class Clerk : public EmployeeBase {
```

```
public:
    std::string task;

    Clerk(std::string n, int id, std::string t) : EmployeeBase(n, id), task(t) {}

    void displayClerkInfo() const {
        displayBaseInfo();
        std::cout << ", Task: " << task << std::endl;
    }
};

void solveHierarchicalInheritance() {
    std::cout << "\n--- 13. Hierarchical Inheritance ---" << std::endl;
    Manager m("Bob Johnson", 101, "Sales");
    Clerk cl("Jane Doe", 102, "Filing");
    m.displayManagerInfo();
    cl.displayClerkInfo();
}

// 14. Hybrid Inheritance (Medium)
// Problem: Create a class structure to demonstrate hybrid inheritance using Student, Sports, and Marks classes, all contributing to a Result class.
class StudentInfo {
public:
    std::string name;
    int rollNumber;

    StudentInfo(std::string n, int roll) : name(n), rollNumber(roll) {}

    void displayStudentInfo() const {
        std::cout << "Name: " << name << ", Roll No: " << rollNumber;
```

```
    }
};

class Sports {
public:
    std::string sportName;
    int score;

    Sports(std::string s, int sc) : sportName(s), score(sc) {}

    void displaySportsInfo() const {
        std::cout << ", Sport: " << sportName << ", Score: " << score;
    }
};

class Marks {
public:
    int mathMarks;
    int scienceMarks;

    Marks(int math, int science) : mathMarks(math), scienceMarks(science) {}

    void displayMarks() const {
        std::cout << ", Math: " << mathMarks << ", Science: " << scienceMarks;
    }
};

class Result : public StudentInfo, public Sports, public Marks {
public:
    double totalMarks;
```

```
    Result(std::string n, int roll, std::string s, int sc, int math, int science)
        : StudentInfo(n, roll), Sports(s, sc), Marks(math, science), totalMarks(math + science) {}

    void displayResult() const {
        displayStudentInfo();
        displaySportsInfo();
        displayMarks();
        std::cout << ", Total Marks: " << totalMarks << std::endl;
    }
};

void solveHybridInheritance() {
    std::cout << "\n--- 14. Hybrid Inheritance ---" << std::endl;
    Result r("Charlie", 201, "Football", 90, 85, 92);
    r.displayResult();
}

// 15. Composition ("Has-a" relationship) (Easy)
// Problem: Create a class Engine and a class Car that has an Engine object. Show composition by calling a function of Engine through Car.
class Engine {
public:
    void start() const {
        std::cout << "Engine started." << std::endl;
    }
};

class CarWithEngine {
public:
    Engine carEngine; // Composition
```

```
    void startCar() const {
        carEngine.start();
        std::cout << "Car is running." << std::endl;
    }
};

void solveComposition() {
    std::cout << "\n--- 15. Composition (\"Has-a\" relationship) ---" << std::endl;
    CarWithEngine myCar;
    myCar.startCar();
}

#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <fstream>
#include <memory> // For smart pointers

// ... (Previous 15 problems' code remains here) ...

// 16. Static Members (Easy)
// Problem: Write a class Counter that counts the number of objects created using a static member.
class Counter {
public:
    static int count;

    Counter() {
        count++;
    }
```

```
    static void displayCount() {
        std::cout << "Number of objects created: " << count << std::endl;
    }
};
int Counter::count = 0; // Initialize the static member

void solveStaticMembers() {
    std::cout << "\n--- 16. Static Members ---" << std::endl;
    Counter c1;
    Counter c2;
    Counter::displayCount();
    Counter c3;
    Counter::displayCount();
}

// 17. Friend Function (Medium)
// Problem: Create a class Box and use a friend function compareVolume() to compare volumes of two Box objects.
class BoxWithFriend {
private:
    double width;
    double height;
    double depth;

public:
    BoxWithFriend(double w, double h, double d) : width(w), height(h), depth(d) {}

    double getVolume() const {
        return width * height * depth;
    }
```

```
    friend bool compareVolume(const BoxWithFriend& box1, const BoxWithFriend& box2);
};

bool compareVolume(const BoxWithFriend& box1, const BoxWithFriend& box2) {
    return box1.getVolume() > box2.getVolume();
}

void solveFriendFunction() {
    std::cout << "\n--- 17. Friend Function ---" << std::endl;
    BoxWithFriend box1(2.0, 3.0, 4.0);
    BoxWithFriend box2(3.0, 3.0, 3.0);

    if (compareVolume(box1, box2)) {
        std::cout << "Box 1 has a larger volume than Box 2." << std::endl;
    } else {
        std::cout << "Box 2 has a larger or equal volume to Box 1." << std::endl;
    }
}

// 18. Inline Functions (Easy)
// Problem: Define an inline member function in class Math that returns the square of a number.
class Math {
public:
    inline int square(int num) {
        return num * num;
    }
};

void solveInlineFunctions() {
    std::cout << "\n--- 18. Inline Functions ---" << std::endl;
```

```
    Math m;
    std::cout << "Square of 5: " << m.square(5) << std::endl;
}

// 19. Default Constructor and Destructor (Easy)
// Problem: Create a class Demo that displays messages from its constructor and destructor.
class Demo {
public:
    Demo() {
        std::cout << "Constructor called for Demo object." << std::endl;
    }

    ~Demo() {
        std::cout << "Destructor called for Demo object." << std::endl;
    }
};

void solveDefaultConstructorDestructor() {
    std::cout << "\n--- 19. Default Constructor and Destructor ---" << std::endl;
    {
        Demo d; // Object created, constructor called
    }       // Object goes out of scope, destructor called
}

// 20. Virtual Function and Runtime Polymorphism (Medium)
// Problem: Create a base class Account with a virtual function calculateInterest(), and override
it in SavingsAccount and CurrentAccount.
class Account {
public:
    virtual void calculateInterest() {
        std::cout << "Calculating interest for a generic account." << std::endl;
```

```
    }
    virtual ~Account() {}
};

class SavingsAccount : public Account {
public:
    void calculateInterest() override {
        std::cout << "Calculating interest for a savings account." << std::endl;
    }
};

class CurrentAccount : public Account {
public:
    void calculateInterest() override {
        std::cout << "Calculating interest for a current account." << std::endl;
    }
};

void solveVirtualFunctionPolymorphism() {
    std::cout << "\n--- 20. Virtual Function and Runtime Polymorphism ---" << std::endl;
    Account* acc1 = new SavingsAccount();
    Account* acc2 = new CurrentAccount();

    acc1->calculateInterest(); // Calls SavingsAccount::calculateInterest()
    acc2->calculateInterest(); // Calls CurrentAccount::calculateInterest()

    delete acc1;
    delete acc2;
}

// 21. Array of Objects (Easy)
```

```
// Problem: Create a class Item with properties id and price. Create an array of 5 objects and display their details.
class Item {
public:
  int id;
  double price;

  void displayItem() const {
    std::cout << "ID: " << id << ", Price: " << price << std::endl;
  }
};

void solveArrayOfObjects() {
  std::cout << "\n--- 21. Array of Objects ---" << std::endl;
  Item items[5];
  items[0].id = 1; items[0].price = 10.50;
  items[1].id = 2; items[1].price = 20.75;
  items[2].id = 3; items[2].price = 5.99;
  items[3].id = 4; items[3].price = 15.20;
  items[4].id = 5; items[4].price = 30.00;

  for (int i = 0; i < 5; ++i) {
    items[i].displayItem();
  }
}

// 22. Multiple Inheritance (Medium)
// Problem: Create classes Teacher and Researcher. Derive a class Professor that inherits from both. Show multiple inheritance in action.
class Teacher {
public:
  std::string subject;
```

```
    void displayTeacherInfo() const {
        std::cout << "Subject: " << subject;
    }
};

class Researcher {
public:
    std::string researchArea;

    void displayResearcherInfo() const {
        std::cout << ", Research Area: " << researchArea;
    }
};

class Professor : public Teacher, public Researcher {
public:
    std::string department;

    void displayProfessorInfo() const {
        std::cout << "Department: " << department << ", ";
        displayTeacherInfo();
        displayResearcherInfo();
        std::cout << std::endl;
    }
};

void solveMultipleInheritance() {
    std::cout << "\n--- 22. Multiple Inheritance ---" << std::endl;
    Professor prof;
    prof.subject = "Computer Science";
```

```
    prof.researchArea = "Artificial Intelligence";
    prof.department = "Faculty of Engineering";
    prof.displayProfessorInfo();
}

// 23. Operator Overloading: Unary Operator (Medium)
// Problem: Overload the unary - operator to negate the value of a class object representing a number.
class Number {
private:
    int value;

public:
    Number(int v) : value(v) {}

    Number operator-() const {
        return Number(-value);
    }

    void display() const {
        std::cout << "Value: " << value << std::endl;
    }
};

void solveUnaryOperatorOverloading() {
    std::cout << "\n--- 23. Operator Overloading: Unary Operator ---" << std::endl;
    Number num1(10);
    Number num2 = -num1;
    num1.display();
    num2.display();
}
```

```
// 24. Copy Constructor (Medium)
// Problem: Create a class Book and implement a copy constructor that copies the details of another book object.
class BookWithCopy {
public:
    std::string title;
    double price;

    BookWithCopy(std::string t, double p) : title(t), price(p) {}

    BookWithCopy(const BookWithCopy& other) : title(other.title), price(other.price) {
        std::cout << "Copy constructor called." << std::endl;
    }

    void displayDetails() const {
        std::cout << "Title: " << title << ", Price: " << price << std::endl;
    }
};

void solveCopyConstructor() {
    std::cout << "\n--- 24. Copy Constructor ---" << std::endl;
    BookWithCopy book1("Effective C++", 35.50);
    BookWithCopy book2 = book1; // Copy constructor called
    book1.displayDetails();
    book2.displayDetails();
}

// 25. Constructor with Default Arguments (Easy)
// Problem: Create a class Rectangle with a constructor that has default values for width and height.
class RectangleWithDefault {
```

```
public:
    double width;
    double height;

    RectangleWithDefault(double w = 1.0, double h = 1.0) : width(w), height(h) {}

    void displayDimensions() const {
        std::cout << "Width: " << width << ", Height: " << height << std::endl;
    }
};

void solveConstructorWithDefaultArguments() {
    std::cout << "\n--- 25. Constructor with Default Arguments ---" << std::endl;
    RectangleWithDefault rect1;
    RectangleWithDefault rect2(5.0);
    RectangleWithDefault rect3(3.0, 7.0);

    rect1.displayDimensions();
    rect2.displayDimensions();
    rect3.displayDimensions();
}

// 26. Virtual Destructor (Medium)
// Problem: Create a base class Shape and derived class Triangle. Use a virtual destructor to ensure proper cleanup.
class ShapeWithVirtualDestructor {
public:
    virtual ~ShapeWithVirtualDestructor() {
        std::cout << "Shape destructor called." << std::endl;
    }
};
```

```
class Triangle : public ShapeWithVirtualDestructor {
public:
    Triangle() {
        std::cout << "Triangle constructor called." << std::endl;
    }
    ~Triangle() override {
        std::cout << "Triangle destructor called." << std::endl;
    }
};

void solveVirtualDestructor() {
    std::cout << "\n--- 26. Virtual Destructor ---" << std::endl;
    ShapeWithVirtualDestructor* shapePtr = new Triangle();
    delete shapePtr; // Virtual destructor ensures Triangle's destructor is called
}

// 27. Overloading Comparison Operator (Medium)
// Problem: Overload the == operator in a Time class to compare two time objects.
class Time {
private:
    int hours;
    int minutes;
    int seconds;

public:
    Time(int h = 0, int m = 0, int s = 0) : hours(h), minutes(m), seconds(s) {}

    bool operator==(const Time& other) const {
        return (hours == other.hours && minutes == other.minutes && seconds == other.seconds);
    }
```

```
    void display() const {
        std::cout << hours << ":" << minutes << ":" << seconds << std::endl;
    }
};

void solveOverloadingComparisonOperator() {
    std::cout << "\n--- 27. Overloading Comparison Operator ---" << std::endl;
    Time t1(10, 30, 45);
    Time t2(10, 30, 45);
    Time t3(11, 0, 0);

    if (t1 == t2) {
        std::cout << "Time t1 and t2 are equal." << std::endl;
    } else {
        std::cout << "Time t1 and t2 are not equal." << std::endl;
    }

    if (t1 == t3) {
        std::cout << "Time t1 and t3 are equal." << std::endl;
    } else {
        std::cout << "Time t1 and t3 are not equal." << std::endl;
    }
}

// 28. File Handling with Class (Medium)
// Problem: Create a class Student and write object data to a file and read it back.
class StudentFile {
public:
    std::string name;
    int rollNumber;
```

```
    StudentFile(std::string n = "", int roll = 0) : name(n), rollNumber(roll) {}

    void writeToFile(const std::string& filename) const {
        std::ofstream outfile(filename, std::ios::binary);
        outfile.write(reinterpret_cast<const char*>(this), sizeof(*this));
        outfile.close();
        std::cout << "Student data written to " << filename << std::endl;
    }

    void readFromFile(const std::string& filename) {
        std::ifstream infile(filename, std::ios::binary);
        if (infile.read(reinterpret_cast<char*>(this), sizeof(*this))) {
            std::cout << "Student data read from " << filename << std::endl;
            displayDetails();
        } else {
            std::cerr << "Error reading from file or file is empty." << std::endl;
        }
        infile.close();
    }

    void displayDetails() const {
        std::cout << "Name: " << name << ", Roll No: " << rollNumber << std::endl;
    }
};

void solveFileHandlingWithClass() {
    std::cout << "\n--- 28. File Handling with Class ---" << std::endl;
    StudentFile student1("Eva", 110);
    student1.writeToFile("student_data.bin");
```

```
    StudentFile student2;
    student2.readFromFile("student_data.bin");
}

// 29. Object as Function Argument (Easy)
// Problem: Create a class Circle and write a function that takes a Circle object as an argument to compute area.
class CircleForArea {
private:
    double radius;

public:
    CircleForArea(double r) : radius(r) {}

    double getRadius() const {
        return radius;
    }
};

double computeArea(const CircleForArea& circle) {
    return M_PI * circle.getRadius() * circle.getRadius();
}

void solveObjectAsFunctionArgument() {
    std::cout << "\n--- 29. Object as Function Argument ---" << std::endl;
    CircleForArea c(5.0);
    double area = computeArea(c);
    std::cout << "Area of the circle with radius " << c.getRadius() << " is: " << area << std::endl;
}

// 30. Array of Pointers to Objects (Medium)
```

```
// Problem: Create a class Animal and an array of pointers to dynamically allocate and manage 3 animals.
class AnimalPtr {
public:
  virtual void speak() const {
    std::cout << "Generic animal sound." << std::endl;
  }
  virtual ~AnimalPtr() {}
};

class DogPtr : public AnimalPtr {
public:
  void speak() const override {
    std::cout << "Woof!" << std::endl;
  }
};

class CatPtr : public AnimalPtr {
public:
  void speak() const override {
    std::cout << "Meow!" << std::endl;
  }
};

void solveArrayOfPointersToObject() {
  std::cout << "\n--- 30. Array of Pointers to Objects ---" << std::endl;
  AnimalPtr* animals[3];
  animals[0] = new DogPtr();
  animals[1] = new CatPtr();
  animals[2] = new AnimalPtr();
```

```
    for (int i = 0; i < 3; ++i) {
        animals[i]->speak();
        delete animals[i]; // Remember to deallocate memory
    }
}

int main() {
    solveClassAndObjectBasics();
    solveConstructorInitialization();
    solveSingleInheritance();
    solveMultilevelInheritance();
    solveMethodOverloading();
    solveMethodOverridingPolymorphism();
    solveOperatorOverloading();
    solveAbstractClass();
    solveConstructorOverloading();
    solveExceptionHandling();
    solveEncapsulation();
    solveProtectedAccessModifier();
    solveHierarchicalInheritance();
    solveHybridInheritance();
    solveComposition();
    solveStaticMembers();
    solveFriendFunction();
    solveInlineFunctions();
    solveDefaultConstructorDestructor();
    solveVirtualFunctionPolymorphism();
    solveArrayOfObjects();
    solveMultipleInheritance();
    solveUnaryOperatorOverloading();
    solveCopyConstructor();
```

```
    solveConstructorWithDefaultArguments();
    solveVirtualDestructor();
    solveOverloadingComparisonOperator();
    solveFileHandlingWithClass();
    solveObjectAsFunctionArgument();
    solveArrayOfPointersToObject();

    return 0;
}
```