

OBJECT ORIENTED PROGRAMMING (C++)

Chapter 4: Inheritance

Coding Problems and Solutions - Set 2

This is a second set of lab/exam-style programming problems on Inheritance, covering scenarios not included in the first set - function overriding, runtime polymorphism with pointer arrays, default arguments in constructors, nested classes, combined IS-A/HAS-A design, and friend-based operator overloading.

Problem 1: Function overriding in single inheritance

Problem Statement: Write a program where a base class Shape defines a draw() function, and a derived class Circle overrides it with its own implementation.

```
#include <iostream>
using namespace std;

class Shape
{
public:
    void draw() { cout << "Drawing a generic shape." << endl; }
};

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

int main()
{
    Shape s;
    Circle c;
    s.draw();
    c.draw();          // calls Circle's own draw()
    c.Shape::draw();    // explicitly calls the base version
    return 0;
}
```

Output: Drawing a generic shape. | Drawing a circle. | Drawing a generic shape.

Explanation: Circle defines a function with the same name and signature as Shape's draw(), which hides (overrides) the base version for Circle objects. The base version is not lost - it can still be reached explicitly using the scope resolution operator Shape::draw().

Problem 2: Protected member access across a 3-level chain

Problem Statement: Write a program with a 3-level inheritance chain BankAccount -> SavingsAccount -> FixedDeposit, where each level adds a feature while directly using the protected balance inherited from the top.

```

#include <iostream>
using namespace std;

class BankAccount
{
protected:
    float balance;
public:
    BankAccount(float b) : balance(b) {}
};

class SavingsAccount : public BankAccount
{
protected:
    float interestRate;
public:
    SavingsAccount(float b, float r) : BankAccount(b), interestRate(r) {}
    void addInterest() { balance += balance * interestRate / 100; }
};

class FixedDeposit : public SavingsAccount
{
public:
    int lockInYears;
    FixedDeposit(float b, float r, int y) : SavingsAccount(b, r), lockInYears(y) {}
    void showMaturity()
    {
        addInterest();
        cout << "Maturity balance after " << lockInYears << " year(s): " << balance <<
endl;
    }
};

int main()
{
    FixedDeposit fd(10000, 8, 1);
    fd.showMaturity();
    return 0;
}

```

Output: Maturity balance after 1 year(s): 10800

Explanation: balance, declared protected in BankAccount, remains directly accessible two levels down in FixedDeposit because protected access is preserved across public inheritance at every level of the chain, unlike private members which would not be.

Problem 3: Multiple inheritance combining two unrelated data sources

Problem Statement: Write a program where a class Result inherits from two unrelated classes, Marks and Attendance, and combines their data (not just function names) to decide eligibility.

```

#include <iostream>
using namespace std;

class Marks
{
protected:
    int score;
public:
    Marks(int s) : score(s) {}
};

```

```

class Attendance
{
protected:
    int percentage;
public:
    Attendance(int p) : percentage(p) {}
};

class Result : public Marks, public Attendance
{
public:
    Result(int s, int p) : Marks(s), Attendance(p) {}
    void checkEligibility()
    {
        if (score >= 40 && percentage >= 75)
            cout << "Eligible to appear for exam." << endl;
        else
            cout << "Not eligible." << endl;
    }
};

int main()
{
    Result r(65, 80);
    r.checkEligibility();
    return 0;
}

```

Output: Eligible to appear for exam.

Explanation: Result inherits data members from two independent base classes, Marks and Attendance, and both must be passed through Result's own constructor to their respective base constructors. checkEligibility() then freely combines score and percentage, showing how multiple inheritance can merge data (not just behaviour) from unrelated sources.

Problem 4: Hierarchical inheritance with virtual functions and an array of base pointers

Problem Statement: Write a program where classes Car and Bike are derived from a common base class Vehicle with a virtual function fuelType(), and use an array of Vehicle pointers to call the correct version for each object.

```

#include <iostream>
using namespace std;

class Vehicle
{
public:
    virtual void fuelType() { cout << "Generic vehicle fuel." << endl; }
};

class Car : public Vehicle
{
public:
    void fuelType() override { cout << "Car runs on Petrol/Diesel." << endl; }
};

class Bike : public Vehicle
{
public:

```

```

    void fuelType() override { cout << "Bike runs on Petrol." << endl; }
};

int main()
{
    Vehicle* vehicles[2];
    vehicles[0] = new Car();
    vehicles[1] = new Bike();
    for (int i = 0; i < 2; i++)
        vehicles[i]->fuelType();    // resolved at runtime for each object
    return 0;
}

```

Output: Car runs on Petrol/Diesel. | Bike runs on Petrol.

Explanation: Car and Bike both inherit independently from Vehicle (hierarchical inheritance) and override its virtual function fuelType(). Because fuelType() is virtual, calling it through a Vehicle pointer still invokes each object's own overridden version - runtime polymorphism working together with inheritance.

Problem 5: Abstract class with an array of derived-class pointers to total areas

Problem Statement: Write a program using an abstract class Shape with a pure virtual area() function, store different derived shape objects in an array of base class pointers, and print their total area.

```

#include <iostream>
using namespace std;

class Shape
{
public:
    virtual double area() = 0;
};

class Square : public Shape
{
    double side;
public:
    Square(double s) : side(s) {}
    double area() override { return side * side; }
};

class Triangle : public Shape
{
    double base, height;
public:
    Triangle(double b, double h) : base(b), height(h) {}
    double area() override { return 0.5 * base * height; }
};

int main()
{
    Shape* shapes[2] = { new Square(4), new Triangle(6, 5) };
    double total = 0;
    for (int i = 0; i < 2; i++) total += shapes[i]->area();
    cout << "Total area: " << total << endl;
    return 0;
}

```

Output: Total area: 31

Explanation: Since Shape has a pure virtual area(), it cannot be instantiated, but an array of Shape pointers can still store addresses of concrete derived objects (Square, Triangle). Looping through the array and calling ->area() automatically invokes each object's own version, letting the loop total areas of different shape types uniformly.

Problem 6: Default arguments in a derived class constructor

Problem Statement: Write a program showing a derived class constructor using default arguments so that a Rectangle object can be created either as a plain rectangle or, using the same constructor, as a square.

```
#include <iostream>
using namespace std;

class Shape
{
protected:
    string name;
public:
    Shape(string n) : name(n) {}
};

class Rectangle : public Shape
{
    double length, breadth;
public:
    Rectangle(double l, double b = -1) : Shape("Rectangle"), length(l)
    {
        breadth = (b == -1) ? l : b;    // if breadth not given, treat as a square
    }
    void show() { cout << name << " area: " << length * breadth << endl; }
};

int main()
{
    Rectangle r1(5, 3);    // proper rectangle
    Rectangle r2(4);        // acts as a square (4 x 4)
    r1.show();
    r2.show();
    return 0;
}
```

Output: Rectangle area: 15 | Rectangle area: 16

Explanation: The derived class constructor Rectangle(double l, double b = -1) uses a default argument, so callers may omit breadth entirely. The constructor still explicitly initializes the base class through Shape("Rectangle") regardless of how many arguments were supplied, showing default arguments and base-class initialization working together.

Problem 7: Class within class (nested class) used for a Date field

Problem Statement: Write a program that defines a nested class Date inside an Employee class, and uses it to store an employee's date of joining.

```
#include <iostream>
using namespace std;

class Employee
{

```

```

public:
    class Date    // nested class
    {
        int day, month, year;
    public:
        Date(int d, int m, int y) : day(d), month(m), year(y) {}
        void show() { cout << day << "-" << month << "-" << year; }
    };

    string name;
    Date joiningDate;    // nested class used as a member type
    Employee(string n, int d, int m, int y) : name(n), joiningDate(d, m, y) {}
    void display()
    {
        cout << name << ", Joined on: ";
        joiningDate.show();
        cout << endl;
    }
};

int main()
{
    Employee e("Kabir", 15, 6, 2022);
    e.display();
    return 0;
}

```

Output: Kabir, Joined on: 15-6-2022

Explanation: Date is declared entirely inside Employee, so it is a nested class that exists only to support Employee. It is accessed as Employee::Date outside the class, and Employee holds a Date object (joiningDate) as one of its members, which is also an example of containership using a purpose-built nested type.

Problem 8: Combined IS-A and HAS-A relationships in one program

Problem Statement: Write a program where a Manager IS-A Employee (via inheritance) and every Employee HAS-A Address (via containership), demonstrating both relationships together.

```

#include <iostream>
using namespace std;

class Address
{
    string city;
public:
    Address(string c) : city(c) {}
    void show() { cout << "City: " << city; }
};

class Employee
{
protected:
    string name;
    Address address;    // HAS-A: Employee contains an Address object
public:
    Employee(string n, string c) : name(n), address(c) {}
};

class Manager : public Employee    // IS-A: Manager is a kind of Employee
{
    int teamSize;
}

```

```

public:
    Manager(string n, string c, int t) : Employee(n, c), teamSize(t) {}
    void display()
    {
        cout << name << " manages a team of " << teamSize << ". ";
        address.show();
        cout << endl;
    }
};

int main()
{
    Manager m("Neha", "Siliguri", 6);
    m.display();
    return 0;
}

```

Output: Neha manages a team of 6. City: Siliguri

Explanation: Manager : public Employee models the IS-A relationship (a Manager is a specialised Employee), while the Address object embedded inside Employee models the HAS-A relationship (every Employee has an Address). The Manager class ends up using both relationships at once, which is typical of real-world class designs.

Problem 9: Friend function overloading the + operator for a Distance class

Problem Statement: Write a program using a friend function to overload the + operator so that two Distance objects (stored in feet and inches) can be added directly.

```

#include <iostream>
using namespace std;

class Distance
{
    int feet, inches;
public:
    Distance(int f, int i) : feet(f), inches(i) {}
    friend Distance operator+(Distance d1, Distance d2);
    void show() { cout << feet << "'" << inches << "'" << endl; }
};

Distance operator+(Distance d1, Distance d2)
{
    int totalInches = (d1.feet * 12 + d1.inches) + (d2.feet * 12 + d2.inches);
    return Distance(totalInches / 12, totalInches % 12); // accesses private members directly
}

int main()
{
    Distance d1(5, 8), d2(3, 7);
    Distance d3 = d1 + d2; // invokes the friend operator+ function
    d3.show();
    return 0;
}

```

Output: 9'3"

Explanation: operator+ is declared as a friend of Distance so it can reach the private members feet and inches of both operands without needing getter functions. This is one of the most common real uses of friend functions - implementing binary operator overloads that treat both operands symmetrically.

Problem 10: Two mutually independent friend classes accessing one data class

Problem Statement: Write a program where a single class Sensor has two separate friend classes, DisplayUnit and LoggerUnit, each independently reading Sensor's private reading value.

```
#include <iostream>
using namespace std;

class Sensor
{
    float reading;
    friend class DisplayUnit;
    friend class LoggerUnit;
public:
    Sensor(float r) : reading(r) {}
};

class DisplayUnit
{
public:
    void show(Sensor s) { cout << "Display -> Reading: " << s.reading << endl; }
};

class LoggerUnit
{
public:
    void log(Sensor s) { cout << "Log -> Recorded value: " << s.reading << endl; }
};

int main()
{
    Sensor temp(36.7f);
    DisplayUnit d;
    LoggerUnit l;
    d.show(temp);
    l.log(temp);
    return 0;
}
```

Output: Display -> Reading: 36.7 | Log -> Recorded value: 36.7

Explanation: Sensor grants friendship to two separate classes independently. Both DisplayUnit and LoggerUnit can read Sensor's private reading, but neither of them is a friend of the other, and Sensor itself has no special access to either of them - friendship here is one-directional and granted individually to each class.