

OBJECT ORIENTED PROGRAMMING (C++)

Chapter 4: Inheritance

Coding Problems and Solutions

This booklet contains lab/exam-style programming problems based on Inheritance, each with a complete, compilable C++ solution, sample output, and a short explanation of the underlying concept used.

Problem 1: Single inheritance - Employee salary calculation

Problem Statement: Write a C++ program using single inheritance, where a class Employee stores name and basic salary, and a derived class Manager adds a bonus amount and computes total salary.

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

class Employee
{
protected:
    string name;
    float basicSalary;
public:
    void getBasicData(string n, float s) { name = n; basicSalary = s; }
};

class Manager : public Employee
{
    float bonus;
public:
    void getBonus(float b) { bonus = b; }
    void showSalary()
    {
        cout << "Name: " << name << endl;
        cout << "Total Salary: " << (basicSalary + bonus) << endl;
    }
};

int main()
{
    Manager m;
    m.getBasicData("Sourav", 35000);
    m.getBonus(5000);
    m.showSalary();
    return 0;
}
```

Output: Name: Sourav | Total Salary: 40000

Explanation: Manager inherits publicly from Employee, so it can directly use the protected members name and basicSalary inside showSalary() while adding its own member bonus. This is a straightforward application of single inheritance for code reuse.

Problem 2: Multilevel inheritance - Animal to Mammal to Dog

Problem Statement: Write a program demonstrating multilevel inheritance with three classes: Animal (base), Mammal (derived from Animal), and Dog (derived from Mammal).

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

class Animal
{
public:
    void eat() { cout << "This animal eats food." << endl; }
};

class Mammal : public Animal
{
public:
    void walk() { cout << "This mammal walks on land." << endl; }
};

class Dog : public Mammal
{
public:
    void bark() { cout << "The dog barks." << endl; }
};

int main()
{
    Dog d;
    d.eat();    // inherited from Animal (level 1)
    d.walk();   // inherited from Mammal (level 2)
    d.bark();   // Dog's own member
    return 0;
}
```

Output: This animal eats food. | This mammal walks on land. | The dog barks.

Explanation: Dog inherits from Mammal, and Mammal inherits from Animal, forming a chain (Animal -> Mammal -> Dog). A Dog object therefore has access to members from all classes in the chain, demonstrating multilevel inheritance.

Problem 3: Hierarchical inheritance - common base class Shape

Problem Statement: Write a program demonstrating hierarchical inheritance, where classes Circle and Rectangle are both derived independently from a common base class Shape.

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

class Shape
{
protected:
    string color = "Black";
public:
    void showColor() { cout << "Color: " << color << endl; }
};

class Circle : public Shape
{
    float radius;
public:
    Circle(float r) : radius(r) {}
}
```

```

    void area() { cout << "Circle area: " << 3.14159 * radius * radius << endl; }
};

class Rectangle : public Shape
{
    float l, b;
public:
    Rectangle(float l1, float b1) : l(l1), b(b1) {}
    void area() { cout << "Rectangle area: " << l * b << endl; }
};

int main()
{
    Circle c(5);
    Rectangle r(4, 6);
    c.showColor(); c.area();
    r.showColor(); r.area();
    return 0;
}

```

Output: Color: Black | Circle area: 78.5397 | Color: Black | Rectangle area: 24

Explanation: Both Circle and Rectangle inherit independently from the single base class Shape, each reusing showColor() while defining their own area() logic. This is hierarchical inheritance - one base, many independent derived classes.

Problem 4: Multiple inheritance with ambiguity resolved by redefinition

Problem Statement: Write a program where a class Result inherits from two classes Sports and Academic, both of which define a function called grade(). Resolve the resulting ambiguity by redefining grade() inside Result.

```

#include <iostream>
using namespace std;

class Sports
{
public:
    void grade() { cout << "Sports Grade: A" << endl; }
};

class Academic
{
public:
    void grade() { cout << "Academic Grade: A+" << endl; }
};

class Result : public Sports, public Academic
{
public:
    void grade()    // redefined here to remove ambiguity
    {
        Sports::grade();
        Academic::grade();
        cout << "Overall Result: PASS" << endl;
    }
};

int main()
{
    Result r;
}

```

```
r.grade(); // calls Result's own grade(), no ambiguity
return 0;
}
```

Output: Sports Grade: A | Academic Grade: A+ | Overall Result: PASS

Explanation: Since both Sports and Academic define grade(), calling r.grade() directly would be ambiguous. Redefining grade() in Result gives it a version of its own, which takes precedence; inside it, the scope resolution operator is used to explicitly call each base class's version.

Problem 5: Hybrid inheritance and the diamond problem using virtual base class

Problem Statement: Write a program to demonstrate hybrid inheritance (a mix of hierarchical and multiple inheritance) and show how a virtual base class avoids the diamond problem for a class hierarchy Person -> Student, Person -> Player -> SportsStudent.

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

class Person
{
public:
    string name = "Riya";
};

class Student : virtual public Person
{
public:
    void study() { cout << name << " is studying." << endl; }
};

class Player : virtual public Person
{
public:
    void play() { cout << name << " is playing." << endl; }
};

class SportsStudent : public Student, public Player
{
public:
    void show() { cout << "Name: " << name << endl; } // single copy of name
};

int main()
{
    SportsStudent ss;
    ss.study();
    ss.play();
    ss.show();
    return 0;
}
```

Output: Riya is studying. | Riya is playing. | Name: Riya

Explanation: SportsStudent combines hierarchical inheritance (Student and Player both from Person) with multiple inheritance (SportsStudent from both Student and Player) - a hybrid form. Without the virtual keyword, SportsStudent would contain two separate copies of Person's name, making ss.name ambiguous. Declaring Person as a virtual base in both Student and Player ensures only one shared copy exists.

Problem 6: Abstract class and pure virtual functions - payroll system

Problem Statement: Write a program using an abstract class Employee with a pure virtual function calculateSalary(), implemented differently by derived classes FullTimeEmployee and PartTimeEmployee.

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

class Employee
{
public:
    virtual float calculateSalary() = 0;    // pure virtual - makes Employee abstract
};

class FullTimeEmployee : public Employee
{
    float monthlySalary;
public:
    FullTimeEmployee(float s) : monthlySalary(s) {}
    float calculateSalary() override { return monthlySalary; }
};

class PartTimeEmployee : public Employee
{
    float hourlyRate; int hoursWorked;
public:
    PartTimeEmployee(float r, int h) : hourlyRate(r), hoursWorked(h) {}
    float calculateSalary() override { return hourlyRate * hoursWorked; }
};

int main()
{
    Employee* e1 = new FullTimeEmployee(30000);
    Employee* e2 = new PartTimeEmployee(250, 80);
    cout << "Full-time salary: " << e1->calculateSalary() << endl;
    cout << "Part-time salary: " << e2->calculateSalary() << endl;
    return 0;
}
```

Output: Full-time salary: 30000 | Part-time salary: 20000

Explanation: Employee cannot be instantiated directly because calculateSalary() is pure virtual. Each derived class supplies its own implementation, and calling through an Employee pointer invokes the correct version at runtime - a practical use of abstraction and runtime polymorphism together.

Problem 7: Constructors in derived classes with parameter passing

Problem Statement: Write a program to show how a derived class constructor explicitly passes arguments to its base class constructor using a member initializer list.

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

class Account
{
protected:
    string accNo;
public:
    Account(string a) : accNo(a)
    {
```

```

        cout << "Account constructor: " << accNo << endl;
    }
};

class SavingsAccount : public Account
{
    float interestRate;
public:
    SavingsAccount(string a, float r) : Account(a), interestRate(r)
    {
        cout << "SavingsAccount constructor: rate = " << interestRate << endl;
    }
};

int main()
{
    SavingsAccount s("SB1023", 4.5);
    return 0;
}

```

Output: Account constructor: SB1023 | SavingsAccount constructor: rate = 4.5

Explanation: The base class Account has no default constructor, so SavingsAccount must explicitly call Account(a) in its own initializer list. The base constructor always completes before the derived constructor's body runs, which is why the Account line prints first.

Problem 8: Containership - Library containing an array of Book objects

Problem Statement: Write a program to demonstrate containership by creating a Library class that contains an array of Book objects as its data member.

```

#include <iostream>
using namespace std;

class Book
{
    string title;
public:
    void setTitle(string t) { title = t; }
    void show() { cout << "- " << title << endl; }
};

class Library
{
    Book books[3];    // containership: Library HAS-A set of Book objects
public:
    void addBooks()
    {
        books[0].setTitle("C++ Primer");
        books[1].setTitle("Data Structures");
        books[2].setTitle("Operating Systems");
    }
    void showAll()
    {
        cout << "Books in Library:" << endl;
        for (int i = 0; i < 3; i++) books[i].show();
    }
};

int main()
{

```

```
Library lib;
lib.addBooks();
lib.showAll();
return 0;
}
```

Output: Books in Library: | - C++ Primer | - Data Structures | - Operating Systems

Explanation: Library does not inherit from Book; instead it embeds an array of Book objects as a data member, which is containership (a HAS-A relationship). Library's member functions access each Book's public interface using the dot operator.

Problem 9: Namespaces - resolving identically named functions

Problem Statement: Write a program using two namespaces, College and Company, that both define a function called id(), and show how to call each one without ambiguity.

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

namespace College
{
    void id() { cout << "College Roll No: 45" << endl; }
}

namespace Company
{
    void id() { cout << "Employee ID: EMP2024" << endl; }
}

int main()
{
    College::id();
    Company::id();
    return 0;
}
```

Output: College Roll No: 45 | Employee ID: EMP2024

Explanation: Both namespaces define a function with the same name id(), but since each lives in its own namespace, they do not conflict. The scope resolution operator (College:: and Company::) is used to specify exactly which one to call.

Problem 10: Friend function - comparing private data of two objects of the same class

Problem Statement: Write a program using a friend function to compare the private marks of two Student objects and print the one with the higher marks.

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

class Student
{
    string name;
    int marks;
public:
    Student(string n, int m) : name(n), marks(m) {}
    friend void compare(Student s1, Student s2);
};
```

```

void compare(Student s1, Student s2)
{
    if (s1.marks > s2.marks)
        cout << s1.name << " scored higher with " << s1.marks << " marks." << endl;
    else if (s2.marks > s1.marks)
        cout << s2.name << " scored higher with " << s2.marks << " marks." << endl;
    else
        cout << "Both scored equal marks." << endl;
}

int main()
{
    Student a("Arjun", 87), b("Priya", 92);
    compare(a, b);
    return 0;
}

```

Output: Priya scored higher with 92 marks.

Explanation: compare() is not a member function, but because it is declared as a friend of Student, it can directly access the private members name and marks of both objects passed to it, without needing public getter functions.

Problem 11: Friend class - a Auditor class accessing private data of Account

Problem Statement: Write a program where a class Auditor is declared as a friend class of Account, and Auditor's member function directly reads Account's private balance.

```

#include <iostream>
using namespace std;

class Account
{
    float balance;
    friend class Auditor;    // Auditor declared as a friend class
public:
    Account(float b) : balance(b) {}
};

class Auditor
{
public:
    void inspect(Account acc)
    {
        cout << "Audited balance: " << acc.balance << endl;    // direct private access
    }
};

int main()
{
    Account a(75000);
    Auditor au;
    au.inspect(a);
    return 0;
}

```

Output: Audited balance: 75000

Explanation: Because Account declares Auditor as a friend class, every member function of Auditor (here, inspect()) can access Account's private members directly. This is useful when an entire helper/utility class, not just one function, needs privileged access.

Problem 12: Constructor call order in multiple inheritance

Problem Statement: Write a program to verify that in multiple inheritance, base class constructors are executed in the order they are listed in the derived class declaration, not the order in the initializer list.

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

class First
{
public:
    First() { cout << "First constructor" << endl; }
};

class Second
{
public:
    Second() { cout << "Second constructor" << endl; }
};

class Third : public First, public Second    // First listed before Second
{
public:
    Third() : Second(), First()    // initializer list order reversed - ignored
    {
        cout << "Third constructor" << endl;
    }
};

int main()
{
    Third t;
    return 0;
}
```

Output: First constructor | Second constructor | Third constructor

Explanation: Even though the initializer list writes Second() before First(), the compiler still calls First's constructor first because First appears before Second in the base list of 'class Third : public First, public Second'. This confirms that declaration order, not initializer-list order, governs base constructor call sequence.