OBJECT ORIENTED PROGRAMMING

Solutions: Inheritance, Method Overriding, and Constructor Chaining

17 September 2026

---

1. Single Inheritance: Account and SavingsAccount

Problem: Design a base class `Account` and a derived class `SavingsAccount`. The base class must store `accountNumber` and `balance`. The derived class adds `interestRate` and calculates annual interest. Display complete account details.

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

class Account {
protected:
    string accountNumber;
    double balance;
public:
    Account(string accNum, double bal) : accountNumber(accNum), balance(bal) {}

    void displayAccount() const {
        cout << "Account Number: " << accountNumber << "\n";
        cout << "Balance: " << balance << "\n";
    }
};

class SavingsAccount : public Account {
private:
    double interestRate;
public:
    SavingsAccount(string accNum, double bal, double rate)
        : Account(accNum, bal), interestRate(rate) {}

    double calculateInterest() const {
        return balance * interestRate;
    }

    void displayDetails() const {
        displayAccount();
        cout << "Interest Rate: " << (interestRate * 100) << "%\n";
        cout << "Annual Interest: " << calculateInterest() << "\n";
    }
};

int main() {
    SavingsAccount sa("SA-10293", 50000.0, 0.045);
    sa.displayDetails();
    return 0;
}
```

Explanation: Demonstrates single inheritance. The derived class inherits attributes from the parent class and initializes base members through its constructor initialization list.

---

2. Multilevel Inheritance: Person, Student, and GraduateStudent

Problem: Implement a 3-tier multilevel inheritance hierarchy consisting of `Person`, `Student`, and `GraduateStudent`. Forward initialization data through each tier and print all details using `displayProfile()`.

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

class Person {
```

```
protected:
    string name;
    int age;
public:
    Person(string n, int a) : name(move(n)), age(a) {}
};

class Student : public Person {
protected:
    int rollNumber;
    string branch;
public:
    Student(string n, int a, int r, string b)
        : Person(move(n), a), rollNumber(r), branch(move(b)) {}
};

class GraduateStudent : public Student {
private:
    string researchArea;
    string guideName;
public:
    GraduateStudent(string n, int a, int r, string b, string area, string guide)
        : Student(move(n), a, r, move(b)), researchArea(move(area)), guideName(move(guide)
    ) {}

    void displayProfile() const {
        cout << "Name: " << name << "\nAge: " << age << "\nRoll Number: " << rollNumber
             << "\nBranch: " << branch << "\nResearch Area: " << researchArea
             << "\nGuide Name: " << guideName << "\n";
    }
};

int main() {
    GraduateStudent grad("Rahul Verma", 24, 202611, "Artificial Intelligence", "Edge AI",
    "Dr. Sharma");
    grad.displayProfile();
    return 0;
}
```

Explanation: Each derived class invokes its immediate parent constructor in a chain, ensuring state is propagated cleanly upward through all levels.

---

3. Multilevel Inheritance: Vehicle, Car, and ElectricCar

Problem: Implement classes `Vehicle`, `Car`, and `ElectricCar` with attributes added at each tier. Verify ancestral attribute access inside the leaf class.

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

class Vehicle {
protected:
    string brand;
    string model;
public:
    Vehicle(string b, string m) : brand(move(b)), model(move(m)) {}
};

class Car : public Vehicle {
protected:
    int seatingCapacity;
public:
    Car(string b, string m, int seats)
        : Vehicle(move(b), move(m)), seatingCapacity(seats) {}
};
```

```
class ElectricCar : public Car {
private:
    int batteryCapacity;
public:
    ElectricCar(string b, string m, int seats, int battery)
        : Car(move(b), move(m), seats), batteryCapacity(battery) {}

    void displayDetails() const {
        cout << "Brand: " << brand << "\nModel: " << model
             << "\nSeating Capacity: " << seatingCapacity << " seats"
             << "\nBattery Capacity: " << batteryCapacity << " kWh\n";
    }
};

int main() {
    ElectricCar ev("Tata", "Nexon EV", 5, 40);
    ev.displayDetails();
    return 0;
}
```

Explanation: Confirms that attributes marked `protected` remain accessible down the inheritance chain without breaking encapsulation.

---

4. Hierarchical Inheritance: Faculty Payroll System

Problem: Define a base class `Faculty` and two derived classes `Professor` and `AssistantProfessor`. Implement `printPayrollSlip()` calculating distinct allowances for each.

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

class Faculty {
protected:
    int employeeID;
    string name;
    double basicSalary;
public:
    Faculty(int id, string n, double sal)
        : employeeID(id), name(move(n)), basicSalary(sal) {}
};

class Professor : public Faculty {
private:
    int publicationsCount;
public:
    Professor(int id, string n, double sal, int pubs)
        : Faculty(id, move(n), sal), publicationsCount(pubs) {}

    void printPayrollSlip() const {
        double allowance = 0.20 * basicSalary;
        cout << "Professor Slip | ID: " << employeeID << ", Name: " << name
             << ", Publications: " << publicationsCount
             << ", Gross: " << (basicSalary + allowance) << "\n";
    }
};

class AssistantProfessor : public Faculty {
private:
    int labHoursAssigned;
public:
    AssistantProfessor(int id, string n, double sal, int labs)
        : Faculty(id, move(n), sal), labHoursAssigned(labs) {}

    void printPayrollSlip() const {
        double allowance = 0.10 * basicSalary;
        cout << "Assistant Prof Slip | ID: " << employeeID << ", Name: " << name
```

```
             << ", Lab Hours: " << labHoursAssigned
             << ", Gross: " << (basicSalary + allowance) << "\n";
    }
};

int main() {
    Professor p(101, "Dr. Rao", 90000.0, 14);
    AssistantProfessor ap(202, "Prof. Ankit", 55000.0, 8);
    p.printPayrollSlip();
    ap.printPayrollSlip();
    return 0;
}
```

Explanation: Demonstrates hierarchical branching where two distinct concrete types share common state and initialization logic from a single base class.

---

5. Hierarchical Inheritance: Geometric Shapes

Problem: Define a base class `Shape` with `color`, and derive `Circle` and `Rectangle`. Calculate area and display along with color.

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

class Shape {
protected:
    string color;
public:
    Shape(string c) : color(move(c)) {}
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(string c, double r) : Shape(move(c)), radius(r) {}

    void display() const {
        cout << "Circle [Color: " << color << ", Area: " << (3.14159 * radius * radius) <<
    "]\n";
    }
};

class Rectangle : public Shape {
private:
    double length, breadth;
public:
    Rectangle(string c, double l, double b) : Shape(move(c)), length(l), breadth(b) {}

    void display() const {
        cout << "Rectangle [Color: " << color << ", Area: " << (length * breadth) << "]\n"
    ;
    }
};

int main() {
    Circle c("Red", 5.0);
    Rectangle r("Blue", 4.0, 6.0);
    c.display();
    r.display();
    return 0;
}
```

Explanation: Independent subclasses calculate their own geometry while reusing inherited attributes from the shared parent.

6. Multiple Inheritance: AllInOneMachine

Problem: Define two independent classes `Printer` and `Scanner`. Combine both into `AllInOneMachine` using multiple inheritance.

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

class Printer {
public:
    void printDocument(const string& doc) const {
        cout << "Printing document: " << doc << "\n";
    }
};

class Scanner {
public:
    void scanDocument(const string& doc) const {
        cout << "Scanning document: " << doc << "\n";
    }
};

class AllInOneMachine : public Printer, public Scanner {
public:
    void performMaintenance() const {
        cout << "Running machine self-test...\n";
    }
};

int main() {
    AllInOneMachine mfp;
    mfp.printDocument("Lab_Sheet.pdf");
    mfp.scanDocument("Identity_Card.png");
    mfp.performMaintenance();
    return 0;
}
```

Explanation: The derived class inherits operations from two distinct base classes simultaneously without ambiguity.

---

7. Multiple Inheritance: Student Scholarship Evaluation

Problem: Derive `StudentReport` from both `AcademicPerformance` and `SportsPerformance`. Determine scholarship eligibility based on both criteria.

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

class AcademicPerformance {
protected:
    double theoryMarks;
public:
    AcademicPerformance(double marks) : theoryMarks(marks) {}
};

class SportsPerformance {
protected:
    char sportsGrade;
public:
    SportsPerformance(char grade) : sportsGrade(grade) {}
};

class StudentReport : public AcademicPerformance, public SportsPerformance {
```

```
private:
    string studentName;
public:
    StudentReport(string name, double marks, char grade)
        : AcademicPerformance(marks), SportsPerformance(grade), studentName(move(name)) {}

    void generateSummary() const {
        bool eligible = (theoryMarks >= 85.0 && (sportsGrade == 'A' || sportsGrade == 'B')
    );
        cout << "Student: " << studentName << " | Marks: " << theoryMarks
             << " | Grade: " << sportsGrade
             << " | Scholarship: " << (eligible ? "Granted" : "Not Granted") << "\n";
    }
};

int main() {
    StudentReport s1("Aman", 88.5, 'A');
    StudentReport s2("Pooja", 78.0, 'A');
    s1.generateSummary();
    s2.generateSummary();
    return 0;
}
```

Explanation: Shows state aggregation from multiple base classes through multiple inheritance, synthesizing both sources of data inside a member method.

---

8. Method Overriding and Dynamic Dispatch

Problem: Base class `Employee` implements `calculateBonus()`. Derived classes `Manager` and `Intern` override it. Demonstrate runtime polymorphic dispatch.

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

class Employee {
protected:
    double baseSalary;
public:
    Employee(double sal) : baseSalary(sal) {}

    virtual double calculateBonus() const {
        return 0.05 * baseSalary;
    }
    virtual ~Employee() = default;
};

class Manager : public Employee {
public:
    Manager(double sal) : Employee(sal) {}

    double calculateBonus() const override {
        return 0.15 * baseSalary;
    }
};

class Intern : public Employee {
public:
    Intern(double sal) : Employee(sal) {}

    double calculateBonus() const override {
        return 2000.0; // Flat bonus
    }
};

int main() {
    Employee regular(50000.0);
    Manager mgr(80000.0);
```

```
    Intern intern(20000.0);

    Employee* staff[] = { &regular, &mgr, &intern };
    for (const auto* e : staff) {
        cout << "Calculated Bonus: " << e->calculateBonus() << "\n";
    }
    return 0;
}
```

Explanation: Demonstrates runtime polymorphism using `virtual` and `override`. Calling the function through base pointers triggers dynamic dispatch to the actual object type.

---

9. Constructor Chaining across Inheritance Boundaries

Problem: Implement constructor chaining from derived class `HeavyBox` to base class `Box` without redundant field assignments.

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

class Box {
protected:
    double width, height, depth;
public:
    Box(double w, double h, double d) : width(w), height(h), depth(d) {}

    void printDimensions() const {
        cout << "Dimensions: " << width << " x " << height << " x " << depth << "\n";
    }
};

class HeavyBox : public Box {
private:
    double weight;
public:
    HeavyBox(double w, double h, double d, double wt)
        : Box(w, h, d), weight(wt) {}

    void display() const {
        printDimensions();
        cout << "Weight: " << weight << " kg\n";
    }
};

int main() {
    HeavyBox box(12.5, 8.0, 15.0, 45.2);
    box.display();
    return 0;
}
```

Explanation: Shows constructor delegation using the initialization list, directly corresponding to `super(w, h, d)` in Java or Python.

---

10. Targeted Base Method Invocation from Overridden Function

Problem: Class `BaseLogger` implements `log()`. Derived class `TimestampLogger` overrides `log()`, appends a timestamp, and explicitly calls `BaseLogger::log()`.

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

class BaseLogger {
public:
    virtual void log(const string& message) const {
```

```
        cout << "[INFO] " << message << "\n";
    }
    virtual ~BaseLogger() = default;
};

class TimestampLogger : public BaseLogger {
public:
    void log(const string& message) const override {
        cout << "[2026-09-17 09:00:00] ";
        BaseLogger::log(message); // Explicit call to base implementation
    }
};

int main() {
    BaseLogger standardLogger;
    TimestampLogger advancedLogger;

    standardLogger.log("Application started.");
    advancedLogger.log("Application started.");
    return 0;
}
```

Explanation: Demonstrates calling overridden base functionality using scope resolution (`BaseLogger::log`) or `super.log()` rather than completely supplanting it.

---

11. Multilevel Chaining and Progressive Overriding: Order Management

Problem: Classes `Order`, `ShippedOrder`, and `DiscountedShippedOrder`. Each derived tier overrides `calculateTotal()` by extending the parent total. Print the charge breakdown.

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

class Order {
protected:
    int orderId;
    string orderDate;
    double baseAmount;
public:
    Order(int id, string date, double amount)
        : orderId(id), orderDate(move(date)), baseAmount(amount) {}

    virtual double calculateTotal() const {
        return baseAmount;
    }
    virtual ~Order() = default;
};

class ShippedOrder : public Order {
protected:
    string trackingNumber;
    double shippingFee;
public:
    ShippedOrder(int id, string date, double amount, string tracking, double fee)
        : Order(id, move(date), amount), trackingNumber(move(tracking)), shippingFee(fee)
    {}

    double calculateTotal() const override {
        return Order::calculateTotal() + shippingFee;
    }
};

class DiscountedShippedOrder : public ShippedOrder {
private:
    double discountPercentage;
public:
```

```
    DiscountedShippedOrder(int id, string date, double amount, string tracking, double fee
    , double discount)
        : ShippedOrder(id, move(date), amount, move(tracking), fee), discountPercentage(
    discount) {}

    double calculateTotal() const override {
        double subtotal = ShippedOrder::calculateTotal();
        return subtotal - (subtotal * (discountPercentage / 100.0));
    }

    void printBreakdown() const {
        cout << "Order ID: " << orderId << " | Date: " << orderDate << "\n";
        cout << "Tracking: " << trackingNumber << "\n";
        cout << "Base Amount: " << baseAmount << "\n";
        cout << "Shipping Fee: +" << shippingFee << "\n";
        cout << "Subtotal: " << ShippedOrder::calculateTotal() << "\n";
        cout << "Discount: -" << discountPercentage << "%\n";
        cout << "Total Payable: " << calculateTotal() << "\n";
    }
};

int main() {
    DiscountedShippedOrder order(5001, "2026-09-17", 4000.0, "TRK987654", 250.0, 10.0);
    order.printBreakdown();
    return 0;
}
```

Explanation: Integrates multilevel inheritance, constructor delegation across 3 tiers, and progressive method overriding where each derived tier reuses the calculation of its direct parent.