Modul 9: OOP - Inheritance

After completing this module, students are expected to:

- Understand the concept of inheritance in OOP

- Implement various types of inheritance in C++

- Use access specifiers in inheritance

- Understand the concept of method overriding

- Apply inheritance for code reusability

1. Basic Concepts of Inheritance

1.1 What is Inheritance?

Inheritance is a mechanism where a class (derived/child class) can inherit properties and methods from another class (base/parent class).

Real-World Analogy: Think of inheritance like family traits:

- A child inherits characteristics from parents (eye color, height)
- The child can also have unique characteristics
- The child can modify inherited traits (different hairstyle)

Benefits of Inheritance:

- **Code Reusability**: Write once, use in multiple classes
- **Logical Hierarchy**: Represents real-world relationships
- **Easier Maintenance**: Changes in base class automatically reflect in derived classes
- **Polymorphism**: Enables treating derived objects as base objects

1.2 Basic Syntax

```
// Base class (Parent)
class BaseClass {
    // members
};

// Derived class (Child)
class DerivedClass : access_specifier BaseClass {
    // additional members
};
```

Example:

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

// Base class
```

```
class Animal {
protected:
    string name;
    int age;

public:
    Animal(string n, int a) : name(n), age(a) {
        cout << "Animal constructor called" << endl;
    }

    void eat() {
        cout << name << " is eating..." << endl;
    }

    void sleep() {
        cout << name << " is sleeping..." << endl;
    }

    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

// Derived class
class Dog : public Animal {
private:
    string breed;

public:
    Dog(string n, int a, string b) : Animal(n, a), breed(b) {
        cout << "Dog constructor called" << endl;
    }

    void bark() {
        cout << name << " is barking: Woof! Woof!" << endl;
    }

    void displayDogInfo() {
        displayInfo();  // Inherited method
        cout << "Breed: " << breed << endl;
```

```
    }
};

int main() {
    Dog myDog("Buddy", 3, "Golden Retriever");

    // Using inherited methods
    myDog.eat();
    myDog.sleep();

    // Using derived class method
    myDog.bark();

    myDog.displayDogInfo();

    return 0;
}
```

Output:

```
Animal constructor called
Dog constructor called
Buddy is eating...
Buddy is sleeping...
Buddy is barking: Woof! Woof!
Name: Buddy, Age: 3
Breed: Golden Retriever
```

1.3 Access Specifiers in Inheritance

Three Types of Inheritance:

Base Class Member	public Inheritance	protected Inheritance	private Inheritance
public members	public	protected	private
protected members	protected	protected	private
private members	Not accessible	Not accessible	Not accessible

Example:

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

class Base {
private:
    int privateVar;

protected:
    int protectedVar;

public:
    int publicVar;

    Base() : privateVar(1), protectedVar(2), publicVar(3) {}
};

// Public Inheritance
class PublicDerived : public Base {
public:
    void access() {
        // privateVar = 10;     // ERROR: private not accessible
        protectedVar = 20;      // OK: protected -> protected
        publicVar = 30;         // OK: public -> public
    }
};

// Protected Inheritance
class ProtectedDerived : protected Base {
public:
    void access() {
        // privateVar = 10;     // ERROR: private not accessible
        protectedVar = 20;      // OK: protected -> protected
        publicVar = 30;         // OK: public -> protected
    }
};

// Private Inheritance
class PrivateDerived : private Base {
public:
    void access() {
```

```
        // privateVar = 10;     // ERROR: private not accessible
        protectedVar = 20;      // OK: protected -> private
        publicVar = 30;         // OK: public -> private
    }
};

int main() {
    PublicDerived pub;
    pub.publicVar = 100;        // OK: public member accessible
    // pub.protectedVar = 200;  // ERROR: protected not accessible outside

    ProtectedDerived prot;
    // prot.publicVar = 100;    // ERROR: public became protected

    PrivateDerived priv;
    // priv.publicVar = 100;    // ERROR: public became private

    return 0;
}
```

Best Practice: Use `public` inheritance for most cases (is-a relationship).

1.4 Constructor and Destructor in Inheritance

Execution Order:

1. **Construction**: Base class constructor → Derived class constructor
2. **Destruction**: Derived class destructor → Base class destructor

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

class Vehicle {
protected:
    string brand;

public:
    Vehicle(string b) : brand(b) {
        cout << "Vehicle constructor: " << brand << endl;
    }
```

```
    ~Vehicle() {
        cout << "Vehicle destructor: " << brand << endl;
    }
};

class Car : public Vehicle {
private:
    string model;

public:
    Car(string b, string m) : Vehicle(b), model(m) {
        cout << "Car constructor: " << model << endl;
    }

    ~Car() {
        cout << "Car destructor: " << model << endl;
    }
};

class ElectricCar : public Car {
private:
    int batteryCapacity;

public:
    ElectricCar(string b, string m, int battery)
        : Car(b, m), batteryCapacity(battery) {
        cout << "ElectricCar constructor: " << batteryCapacity << " kWh" << endl;
    }

    ~ElectricCar() {
        cout << "ElectricCar destructor: " << batteryCapacity << " kWh" << endl;
    }
};

int main() {
    cout << "=== Creating ElectricCar ===" << endl;
    ElectricCar tesla("Tesla", "Model 3", 75);

    cout << "\n=== Destroying ElectricCar ===" << endl;
```

```
    // Destructor called automatically when going out of scope

    return 0;
}
```

Output:

```
=== Creating ElectricCar ===
Vehicle constructor: Tesla
Car constructor: Model 3
ElectricCar constructor: 75 kWh

=== Destroying ElectricCar ===
ElectricCar destructor: 75 kWh
Car destructor: Model 3
Vehicle destructor: Tesla
```

1.5 Practical Example: Employee Hierarchy

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

class Employee {
protected:
    string name;
    int id;
    double baseSalary;

public:
    Employee(string n, int i, double salary)
        : name(n), id(i), baseSalary(salary) {}

    void displayBasicInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
        cout << "Base Salary: $" << baseSalary << endl;
    }

    void work() {
```

```
        cout << name << " is working..." << endl;
    }

    virtual double calculateSalary() {
        return baseSalary;
    }
};

class Manager : public Employee {
private:
    double bonus;
    int teamSize;

public:
    Manager(string n, int i, double salary, double b, int team)
        : Employee(n, i, salary), bonus(b), teamSize(team) {}

    void displayManagerInfo() {
        displayBasicInfo();
        cout << "Bonus: $" << bonus << endl;
        cout << "Team Size: " << teamSize << endl;
    }

    void conductMeeting() {
        cout << name << " is conducting a meeting..." << endl;
    }

    double calculateSalary() override {
        return baseSalary + bonus;
    }
};

class Developer : public Employee {
private:
    string programmingLanguage;
    int projectsCompleted;

public:
    Developer(string n, int i, double salary, string lang, int projects)
        : Employee(n, i, salary), programmingLanguage(lang),
```

```
          projectsCompleted(projects) {}

    void displayDeveloperInfo() {
        displayBasicInfo();
        cout << "Programming Language: " << programmingLanguage << endl;
        cout << "Projects Completed: " << projectsCompleted << endl;
    }

    void writeCode() {
        cout << name << " is writing " << programmingLanguage << " code..." << endl;
    }

    double calculateSalary() override {
        return baseSalary + (projectsCompleted * 500);
    }
};

int main() {
    Manager mgr("Alice Johnson", 101, 8000, 2000, 10);
    Developer dev("Bob Smith", 102, 6000, "C++", 5);

    cout << "=== Manager Information ===" << endl;
    mgr.displayManagerInfo();
    mgr.work();           // Inherited
    mgr.conductMeeting(); // Own method
    cout << "Total Salary: $" << mgr.calculateSalary() << endl;

    cout << "\n=== Developer Information ===" << endl;
    dev.displayDeveloperInfo();
    dev.work();           // Inherited
    dev.writeCode();      // Own method
    cout << "Total Salary: $" << dev.calculateSalary() << endl;

    return 0;
}
```

2. Types of Inheritance and Method Overriding

2.1 Single Inheritance

Definition: One derived class inherits from one base class.

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

class Person {
protected:
    string name;
    int age;

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

    void introduce() {
        cout << "Hi, I'm " << name << ", " << age << " years old." << endl;
    }
};

class Student : public Person {
private:
    string studentId;
    double gpa;

public:
    Student(string n, int a, string id, double g)
        : Person(n, a), studentId(id), gpa(g) {}

    void study() {
        cout << name << " is studying..." << endl;
    }
```

```
    void showGPA() {
        cout << "GPA: " << gpa << endl;
    }
};

int main() {
    Student s("Alice", 20, "S001", 3.8);
    s.introduce();  // Inherited
    s.study();      // Own method
    s.showGPA();    // Own method

    return 0;
}
```

2.2 Multilevel Inheritance

Definition: A class is derived from another derived class.

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

// Level 1: Base class
class LivingBeing {
protected:
    bool isAlive;

public:
    LivingBeing() : isAlive(true) {
        cout << "LivingBeing created" << endl;
    }

    void breathe() {
        cout << "Breathing..." << endl;
    }
};

// Level 2: Derived from LivingBeing
class Animal : public LivingBeing {
```

```
protected:
    string species;

public:
    Animal(string s) : species(s) {
        cout << "Animal created: " << species << endl;
    }

    void move() {
        cout << species << " is moving..." << endl;
    }
};

// Level 3: Derived from Animal
class Dog : public Animal {
private:
    string name;

public:
    Dog(string n) : Animal("Canine"), name(n) {
        cout << "Dog created: " << name << endl;
    }

    void bark() {
        cout << name << " is barking!" << endl;
    }

    void showCapabilities() {
        breathe();  // From LivingBeing
        move();     // From Animal
        bark();     // From Dog
    }
};

int main() {
    Dog myDog("Buddy");
    cout << "\nDog capabilities:" << endl;
    myDog.showCapabilities();

    return 0;
```

```
}
```

Output:

```
LivingBeing created
Animal created: Canine
Dog created: Buddy

Dog capabilities:
Breathing...
Canine is moving...
Buddy is barking!
```

2.3 Multiple Inheritance

Definition: A class inherits from multiple base classes.

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

class Engine {
protected:
    int horsepower;

public:
    Engine(int hp) : horsepower(hp) {
        cout << "Engine: " << horsepower << " HP" << endl;
    }

    void start() {
        cout << "Engine started: " << horsepower << " HP" << endl;
    }
};

class GPS {
protected:
    string currentLocation;

public:
```

```
    GPS(string loc) : currentLocation(loc) {
        cout << "GPS initialized at: " << loc << endl;
    }

    void navigate(string destination) {
        cout << "Navigating from " << currentLocation
             << " to " << destination << endl;
    }
};

class SmartCar : public Engine, public GPS {
private:
    string model;

public:
    SmartCar(string m, int hp, string loc)
        : Engine(hp), GPS(loc), model(m) {
        cout << "SmartCar created: " << model << endl;
    }

    void drive(string destination) {
        cout << "\n=== Driving " << model << " ===" << endl;
        start();            // From Engine
        navigate(destination); // From GPS
        cout << "Arrived at destination!" << endl;
    }
};

int main() {
    SmartCar tesla("Tesla Model S", 670, "New York");
    tesla.drive("Boston");

    return 0;
}
```

Output:

```
Engine: 670 HP
GPS initialized at: New York
SmartCar created: Tesla Model S
```

```
=== Driving Tesla Model S ===
Engine started: 670 HP
Navigating from New York to Boston
Arrived at destination!
```

Diamond Problem in Multiple Inheritance:

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

class Device {
protected:
    int powerConsumption;

public:
    Device(int power) : powerConsumption(power) {
        cout << "Device: " << power << "W" << endl;
    }
};

// Problem: Both inherit from Device
class Printer : public Device {
public:
    Printer(int power) : Device(power) {}
};

class Scanner : public Device {
public:
    Scanner(int power) : Device(power) {}
};

// This creates two copies of Device!
class AllInOne : public Printer, public Scanner {
public:
    AllInOne(int pPower, int sPower)
        : Printer(pPower), Scanner(sPower) {}
    // Now we have ambiguity!
};
```

```
// Solution: Virtual Inheritance
class DeviceVirtual {
protected:
    int powerConsumption;

public:
    DeviceVirtual(int power) : powerConsumption(power) {
        cout << "Device: " << power << "W" << endl;
    }
};

class PrinterVirtual : virtual public DeviceVirtual {
public:
    PrinterVirtual(int power) : DeviceVirtual(power) {}
};

class ScannerVirtual : virtual public DeviceVirtual {
public:
    ScannerVirtual(int power) : DeviceVirtual(power) {}
};

class AllInOneVirtual : public PrinterVirtual, public ScannerVirtual {
public:
    AllInOneVirtual(int power)
        : DeviceVirtual(power), PrinterVirtual(power), ScannerVirtual(power) {}
    // Now only ONE copy of DeviceVirtual
};

int main() {
    AllInOneVirtual device(50);

    return 0;
}
```

2.4 Hierarchical Inheritance

Definition: Multiple derived classes inherit from a single base class.

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

class Shape {
protected:
    string color;

public:
    Shape(string c) : color(c) {}

    void displayColor() {
        cout << "Color: " << color << endl;
    }

    virtual double getArea() = 0;
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(string c, double r) : Shape(c), radius(r) {}

    double getArea() override {
        return 3.14159 * radius * radius;
    }

    void display() {
        cout << "Circle - ";
        displayColor();
        cout << "Radius: " << radius << endl;
        cout << "Area: " << getArea() << endl;
    }
};

class Rectangle : public Shape {
private:
```

```
    double width, height;

public:
    Rectangle(string c, double w, double h)
        : Shape(c), width(w), height(h) {}

    double getArea() override {
        return width * height;
    }

    void display() {
        cout << "Rectangle - ";
        displayColor();
        cout << "Width: " << width << ", Height: " << height << endl;
        cout << "Area: " << getArea() << endl;
    }
};

class Triangle : public Shape {
private:
    double base, height;

public:
    Triangle(string c, double b, double h)
        : Shape(c), base(b), height(h) {}

    double getArea() override {
        return 0.5 * base * height;
    }

    void display() {
        cout << "Triangle - ";
        displayColor();
        cout << "Base: " << base << ", Height: " << height << endl;
        cout << "Area: " << getArea() << endl;
    }
};

int main() {
    Circle circle("Red", 5.0);
```

```
    Rectangle rect("Blue", 4.0, 6.0);
    Triangle tri("Green", 8.0, 5.0);

    circle.display();
    cout << endl;
    rect.display();
    cout << endl;
    tri.display();

    return 0;
}
```

2.5 Method Overriding

Definition: Redefining a base class method in a derived class.

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

class Account {
protected:
    string accountNumber;
    double balance;

public:
    Account(string acc, double bal)
        : accountNumber(acc), balance(bal) {}

    // Method to be overridden
    virtual void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Insufficient funds!" << endl;
        }
    }

    virtual void displayInfo() {
```

```
        cout << "Account: " << accountNumber << endl;
        cout << "Balance: $" << balance << endl;
    }

    double getBalance() { return balance; }
};

class SavingsAccount : public Account {
private:
    double minimumBalance;

public:
    SavingsAccount(string acc, double bal, double minBal)
        : Account(acc, bal), minimumBalance(minBal) {}

    // Override withdraw with additional constraint
    void withdraw(double amount) override {
        if (balance - amount >= minimumBalance) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Cannot withdraw: Minimum balance requirement!" << endl;
            cout << "Minimum balance: $" << minimumBalance << endl;
        }
    }

    void displayInfo() override {
        Account::displayInfo();  // Call base class method
        cout << "Minimum Balance: $" << minimumBalance << endl;
        cout << "Account Type: Savings" << endl;
    }
};

class CheckingAccount : public Account {
private:
    double overdraftLimit;

public:
    CheckingAccount(string acc, double bal, double overdraft)
        : Account(acc, bal), overdraftLimit(overdraft) {}
```

```
    // Override withdraw with overdraft feature
    void withdraw(double amount) override {
        if (balance + overdraftLimit >= amount) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
            if (balance < 0) {
                cout << "Warning: Overdraft used! Balance: $" << balance << endl;
            }
        } else {
            cout << "Cannot withdraw: Exceeds overdraft limit!" << endl;
        }
    }

    void displayInfo() override {
        Account::displayInfo();
        cout << "Overdraft Limit: $" << overdraftLimit << endl;
        cout << "Account Type: Checking" << endl;
    }
};

int main() {
    SavingsAccount savings("SA001", 5000, 1000);
    CheckingAccount checking("CA001", 2000, 500);

    cout << "=== Savings Account ===" << endl;
    savings.displayInfo();
    cout << "\nTrying to withdraw $4500..." << endl;
    savings.withdraw(4500);  // Should fail (below minimum)
    cout << "\nTrying to withdraw $3000..." << endl;
    savings.withdraw(3000);  // Should succeed

    cout << "\n=== Checking Account ===" << endl;
    checking.displayInfo();
    cout << "\nTrying to withdraw $2300..." << endl;
    checking.withdraw(2300);  // Should succeed with overdraft

    return 0;
}
```

3. Practical Applications and Best Practices

3.1 Complete Example: University Management System

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

// Base class
class Person {
protected:
    string name;
    int id;
    int age;

public:
    Person(string n, int i, int a) : name(n), id(i), age(a) {}

    virtual void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
        cout << "Age: " << age << endl;
    }

    virtual string getRole() = 0;  // Pure virtual

    string getName() { return name; }
    int getId() { return id; }

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

```
// Derived class: Student
class Student : public Person {
private:
    string major;
    double gpa;
    vector<string> courses;

public:
    Student(string n, int i, int a, string m, double g)
        : Person(n, i, a), major(m), gpa(g) {}

    void enrollCourse(string course) {
        courses.push_back(course);
        cout << name << " enrolled in " << course << endl;
    }

    void displayInfo() override {
        cout << "=== Student Information ===" << endl;
        Person::displayInfo();
        cout << "Major: " << major << endl;
        cout << "GPA: " << gpa << endl;
        cout << "Enrolled Courses: ";
        if (courses.empty()) {
            cout << "None" << endl;
        } else {
            cout << endl;
            for (const string& course : courses) {
                cout << "  - " << course << endl;
            }
        }
    }

    string getRole() override {
        return "Student";
    }

    void study() {
        cout << name << " is studying..." << endl;
    }
};
```

```
// Derived class: Faculty
class Faculty : public Person {
private:
    string department;
    double salary;
    vector<string> coursesTeaching;

public:
    Faculty(string n, int i, int a, string dept, double sal)
        : Person(n, i, a), department(dept), salary(sal) {}

    void assignCourse(string course) {
        coursesTeaching.push_back(course);
        cout << name << " assigned to teach " << course << endl;
    }

    void displayInfo() override {
        cout << "=== Faculty Information ===" << endl;
        Person::displayInfo();
        cout << "Department: " << department << endl;
        cout << "Salary: $" << salary << endl;
        cout << "Courses Teaching: ";
        if (coursesTeaching.empty()) {
            cout << "None" << endl;
        } else {
            cout << endl;
            for (const string& course : coursesTeaching) {
                cout << "  - " << course << endl;
            }
        }
    }

    string getRole() override {
        return "Faculty";
    }

    void teach() {
        cout << name << " is teaching..." << endl;
    }
```

```
};

// Derived class: Staff
class Staff : public Person {
private:
    string position;
    string department;
    double salary;

public:
    Staff(string n, int i, int a, string pos, string dept, double sal)
        : Person(n, i, a), position(pos), department(dept), salary(sal) {}

    void displayInfo() override {
        cout << "=== Staff Information ===" << endl;
        Person::displayInfo();
        cout << "Position: " << position << endl;
        cout << "Department: " << department << endl;
        cout << "Salary: $" << salary << endl;
    }

    string getRole() override {
        return "Staff";
    }

    void work() {
        cout << name << " (" << position << ") is working..." << endl;
    }
};

// University class to manage all persons
class University {
private:
    string universityName;
    vector<Person*> members;

public:
    University(string name) : universityName(name) {
        cout << "University '" << universityName << "' initialized." << endl;
    }
```

```
void addMember(Person* person) {
    members.push_back(person);
    cout << person->getRole() << " " << person->getName()
         << " added to " << universityName << endl;
}

void displayAllMembers() {
    cout << "\n========== " << universityName << " - All Members ==========" << endl;

    for (Person* member : members) {
        member->displayInfo();
        cout << "----------------------------------------" << endl;
    }
}

void displayByRole(string role) {
    cout << "\n========== " << role << "s at " << universityName << " ==========" << endl;

    bool found = false;
    for (Person* member : members) {
        if (member->getRole() == role) {
            member->displayInfo();
            cout << "----------------------------------------" << endl;
            found = true;
        }
    }

    if (!found) {
        cout << "No " << role << "s found." << endl;
    }
}

Person* findMember(int id) {
    for (Person* member : members) {
        if (member->getId() == id) {
            return member;
        }
    }
    return nullptr;
```

```
    }

    ~University() {
        for (Person* member : members) {
            delete member;
        }
    }
};

int main() {
    University university("Tech University");

    cout << "\n=== Adding Members ===" << endl;

    // Add students
    Student* s1 = new Student("Alice Johnson", 1001, 20, "Computer Science", 3.8);
    Student* s2 = new Student("Bob Smith", 1002, 21, "Electrical Engineering", 3.6);

    university.addMember(s1);
    university.addMember(s2);

    // Add faculty
    Faculty* f1 = new Faculty("Dr. Carol White", 2001, 45, "Computer Science", 85000);
    Faculty* f2 = new Faculty("Dr. David Brown", 2002, 50, "Electrical Engineering", 90000);

    university.addMember(f1);
    university.addMember(f2);

    // Add staff
    Staff* st1 = new Staff("Eve Davis", 3001, 35, "Administrator", "Administration", 50000);

    university.addMember(st1);

    // Assign courses
    cout << "\n=== Assigning Courses ===" << endl;
    f1->assignCourse("Data Structures");
    f1->assignCourse("Algorithms");
    f2->assignCourse("Circuit Analysis");

    // Enroll students
```

```
    cout << "\n=== Enrolling Students ===" << endl;
    s1->enrollCourse("Data Structures");
    s1->enrollCourse("Algorithms");
    s2->enrollCourse("Circuit Analysis");

    // Display all members
    university.displayAllMembers();

    // Display by role
    university.displayByRole("Student");
    university.displayByRole("Faculty");

    // Polymorphic behavior
    cout << "\n=== Daily Activities ===" << endl;
    s1->study();
    f1->teach();
    st1->work();

    return 0;
}
```

3.2 Best Practices

1. Use `virtual` Destructors in Base Classes

```
class Base {
public:
    virtual ~Base() {}  // IMPORTANT for polymorphism
};
```

2. Use `override` Keyword

```
class Derived : public Base {
public:
    void someMethod() override {  // Explicit override
        // Implementation
    }
};
```

3. Prefer Composition Over Inheritance When Appropriate

```
// Don't do: class Car : public Engine
// Do this:
class Car {
private:
    Engine engine;  // Has-a relationship
public:
    Car() : engine() {}
};
```

4. Keep Base Class Interface Stable

```
// Good: Minimal, stable base class
class Shape {
public:
    virtual double getArea() = 0;
    virtual void draw() = 0;
    virtual ~Shape() {}
};
```

5. Use `protected` for Members Needed by Derived Classes

```
class Base {
protected:
    int valueNeededByDerived;  // Accessible in derived classes
private:
    int internalImplementation;  // Not accessible in derived classes
};
```

3.3 Common Pitfalls and Solutions

Problem 1: Forgetting Virtual Destructor

```
// BAD: No virtual destructor
class Base {
public:
    ~Base() { cout << "Base destructor" << endl; }
};

class Derived : public Base {
private:
```

```
    int* data;
public:
    Derived() { data = new int[100]; }
    ~Derived() {
        delete[] data;
        cout << "Derived destructor" << endl;
    }
};

int main() {
    Base* obj = new Derived();
    delete obj;  // MEMORY LEAK! Only Base destructor called
    return 0;
}

// GOOD: Virtual destructor
class Base {
public:
    virtual ~Base() { cout << "Base destructor" << endl; }
};

class Derived : public Base {
private:
    int* data;
public:
    Derived() { data = new int[100]; }
    ~Derived() override {
        delete[] data;
        cout << "Derived destructor" << endl;
    }
};

int main() {
    Base* obj = new Derived();
    delete obj;  // Correct! Both destructors called
    return 0;
}
```

Problem 2: Object Slicing

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

class Animal {
public:
    string type;
    Animal() : type("Animal") {}
    virtual void makeSound() { cout << "Generic sound" << endl; }
};

class Dog : public Animal {
public:
    string breed;
    Dog() : breed("Unknown") { type = "Dog"; }
    void makeSound() override { cout << "Woof!" << endl; }
};

// BAD: Pass by value causes slicing
void processAnimal(Animal animal) {
    animal.makeSound();  // Always calls Animal::makeSound()!
}

// GOOD: Pass by reference or pointer
void processAnimalCorrect(Animal& animal) {
    animal.makeSound();  // Calls correct method
}

void processAnimalPointer(Animal* animal) {
    animal->makeSound();  // Calls correct method
}

int main() {
    Dog myDog;

    cout << "=== Object Slicing (BAD) ===" << endl;
    processAnimal(myDog);  // Slicing occurs! Outputs: Generic sound

    cout << "\n=== Using Reference (GOOD) ===" << endl;
    processAnimalCorrect(myDog);  // Outputs: Woof!
```

```
    cout << "\n=== Using Pointer (GOOD) ===" << endl;
    processAnimalPointer(&myDog);  // Outputs: Woof!

    return 0;
}
```

Problem 3: Calling Virtual Functions in Constructor

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

class Base {
public:
    Base() {
        init();  // Calls Base::init(), not Derived::init()!
    }

    virtual void init() {
        cout << "Base initialization" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        // Base constructor already called
    }

    void init() override {
        cout << "Derived initialization" << endl;
    }
};

int main() {
    Derived d;  // Output: Base initialization (not what we want!)
    return 0;
}

// SOLUTION: Call init() explicitly after construction
```

```
class BaseSolution {
public:
    BaseSolution() {
        // Don't call virtual functions here
    }

    virtual void init() {
        cout << "Base initialization" << endl;
    }
};

class DerivedSolution : public BaseSolution {
public:
    DerivedSolution() {
        // Don't call init() in constructor
    }

    void init() override {
        cout << "Derived initialization" << endl;
    }
};

int main() {
    DerivedSolution d;
    d.init();  // Now calls Derived::init()
    return 0;
}
```

Problem 4: Ambiguity in Multiple Inheritance

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

class ClassA {
public:
    void display() { cout << "ClassA" << endl; }
};

class ClassB {
public:
```

```
    void display() { cout << "ClassB" << endl; }
};

class ClassC : public ClassA, public ClassB {
    // Now ClassC has two display() methods!
};

int main() {
    ClassC obj;
    // obj.display();  // ERROR: Ambiguous! Which display()?

    // SOLUTION 1: Explicitly specify which class
    obj.ClassA::display();
    obj.ClassB::display();

    return 0;
}

// SOLUTION 2: Override in derived class
class ClassCSolution : public ClassA, public ClassB {
public:
    void display() {
        cout << "ClassC - calling both:" << endl;
        ClassA::display();
        ClassB::display();
    }
};
```

3.4 Real-World Example: Vehicle Rental System

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

// Base class
class Vehicle {
protected:
    string vehicleId;
    string brand;
```

```
    string model;
    double dailyRate;
    bool isRented;

public:
    Vehicle(string id, string b, string m, double rate)
        : vehicleId(id), brand(b), model(m), dailyRate(rate), isRented(false) {}

    virtual void displayInfo() {
        cout << "ID: " << vehicleId << endl;
        cout << "Brand: " << brand << endl;
        cout << "Model: " << model << endl;
        cout << "Daily Rate: $" << dailyRate << endl;
        cout << "Status: " << (isRented ? "Rented" : "Available") << endl;
    }

    virtual double calculateRentalCost(int days) {
        return dailyRate * days;
    }

    virtual string getVehicleType() = 0;

    void rent() {
        if (!isRented) {
            isRented = true;
            cout << "Vehicle " << vehicleId << " rented successfully." << endl;
        } else {
            cout << "Vehicle " << vehicleId << " is already rented." << endl;
        }
    }

    void returnVehicle() {
        if (isRented) {
            isRented = false;
            cout << "Vehicle " << vehicleId << " returned successfully." << endl;
        }
    }

    bool checkAvailability() { return !isRented; }
    string getId() { return vehicleId; }
```

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

// Derived class: Car
class Car : public Vehicle {
private:
    int numDoors;
    string fuelType;

public:
    Car(string id, string b, string m, double rate, int doors, string fuel)
        : Vehicle(id, b, m, rate), numDoors(doors), fuelType(fuel) {}

    void displayInfo() override {
        cout << "\n=== CAR ===" << endl;
        Vehicle::displayInfo();
        cout << "Number of Doors: " << numDoors << endl;
        cout << "Fuel Type: " << fuelType << endl;
    }

    double calculateRentalCost(int days) override {
        double baseCost = Vehicle::calculateRentalCost(days);
        // Premium fuel adds 10% to cost
        if (fuelType == "Premium") {
            baseCost *= 1.1;
        }
        return baseCost;
    }

    string getVehicleType() override {
        return "Car";
    }
};

// Derived class: Motorcycle
class Motorcycle : public Vehicle {
private:
    int engineCC;
    bool hasABS;

public:
```

```
    Motorcycle(string id, string b, string m, double rate, int cc, bool abs)
        : Vehicle(id, b, m, rate), engineCC(cc), hasABS(abs) {}

    void displayInfo() override {
        cout << "\n=== MOTORCYCLE ===" << endl;
        Vehicle::displayInfo();
        cout << "Engine CC: " << engineCC << endl;
        cout << "ABS: " << (hasABS ? "Yes" : "No") << endl;
    }

    double calculateRentalCost(int days) override {
        double baseCost = Vehicle::calculateRentalCost(days);
        // Motorcycles get 20% discount for rentals over 7 days
        if (days > 7) {
            baseCost *= 0.8;
        }
        return baseCost;
    }

    string getVehicleType() override {
        return "Motorcycle";
    }
};

// Derived class: Truck
class Truck : public Vehicle {
private:
    double loadCapacity;  // in tons
    bool hasLiftGate;

public:
    Truck(string id, string b, string m, double rate, double capacity, bool liftGate)
        : Vehicle(id, b, m, rate), loadCapacity(capacity), hasLiftGate(liftGate) {}

    void displayInfo() override {
        cout << "\n=== TRUCK ===" << endl;
        Vehicle::displayInfo();
        cout << "Load Capacity: " << loadCapacity << " tons" << endl;
        cout << "Lift Gate: " << (hasLiftGate ? "Yes" : "No") << endl;
    }
```

```
    double calculateRentalCost(int days) override {
        double baseCost = Vehicle::calculateRentalCost(days);
        // Additional charge based on capacity
        baseCost += (loadCapacity * 5 * days);
        // Lift gate adds $10 per day
        if (hasLiftGate) {
            baseCost += (10 * days);
        }
        return baseCost;
    }

    string getVehicleType() override {
        return "Truck";
    }
};

// Rental system manager
class RentalSystem {
private:
    string companyName;
    vector<Vehicle*> fleet;

public:
    RentalSystem(string name) : companyName(name) {
        cout << "=== " << companyName << " Rental System Initialized ===" << endl;
    }

    void addVehicle(Vehicle* vehicle) {
        fleet.push_back(vehicle);
        cout << vehicle->getVehicleType() << " " << vehicle->getId()
             << " added to fleet." << endl;
    }

    void displayFleet() {
        cout << "\n========== " << companyName << " - Fleet ==========" << endl;
        for (Vehicle* vehicle : fleet) {
            vehicle->displayInfo();
            cout << "----------------------------------------" << endl;
        }
    }
```

```
    void displayAvailableVehicles() {
        cout << "\n========== Available Vehicles ==========" << endl;
        bool found = false;
        for (Vehicle* vehicle : fleet) {
            if (vehicle->checkAvailability()) {
                vehicle->displayInfo();
                cout << "----------------------------------------" << endl;
                found = true;
            }
        }
        if (!found) {
            cout << "No vehicles available." << endl;
        }
    }

    Vehicle* findVehicle(string id) {
        for (Vehicle* vehicle : fleet) {
            if (vehicle->getId() == id) {
                return vehicle;
            }
        }
        return nullptr;
    }

    void rentVehicle(string id, int days) {
        Vehicle* vehicle = findVehicle(id);
        if (vehicle != nullptr) {
            if (vehicle->checkAvailability()) {
                vehicle->rent();
                double cost = vehicle->calculateRentalCost(days);
                cout << "Rental Duration: " << days << " days" << endl;
                cout << "Total Cost: $" << cost << endl;
            } else {
                cout << "Vehicle is not available." << endl;
            }
        } else {
            cout << "Vehicle not found." << endl;
        }
    }

    void returnVehicle(string id) {
```

```
        Vehicle* vehicle = findVehicle(id);
        if (vehicle != nullptr) {
            vehicle->returnVehicle();
        } else {
            cout << "Vehicle not found." << endl;
        }
    }

    ~RentalSystem() {
        for (Vehicle* vehicle : fleet) {
            delete vehicle;
        }
    }
};

int main() {
    RentalSystem rental("QuickRent");

    cout << "\n=== Adding Vehicles to Fleet ===" << endl;
    rental.addVehicle(new Car("C001", "Toyota", "Camry", 50, 4, "Regular"));
    rental.addVehicle(new Car("C002", "BMW", "M5", 120, 4, "Premium"));
    rental.addVehicle(new Motorcycle("M001", "Harley", "Sportster", 40, 883, true));
    rental.addVehicle(new Motorcycle("M002", "Honda", "CBR", 35, 600, false));
    rental.addVehicle(new Truck("T001", "Ford", "F-150", 80, 2.5, true));
    rental.addVehicle(new Truck("T002", "Chevrolet", "Silverado", 85, 3.0, false));

    // Display all vehicles
    rental.displayFleet();

    // Rent some vehicles
    cout << "\n=== Renting Vehicles ===" << endl;
    rental.rentVehicle("C001", 3);
    cout << endl;
    rental.rentVehicle("M001", 10);  // Gets discount (>7 days)
    cout << endl;
    rental.rentVehicle("T001", 5);

    // Display available vehicles
    rental.displayAvailableVehicles();

    // Try to rent already rented vehicle
```

```
    cout << "\n=== Trying to Rent Already Rented Vehicle ===" << endl;
    rental.rentVehicle("C001", 2);

    // Return vehicles
    cout << "\n=== Returning Vehicles ===" << endl;
    rental.returnVehicle("C001");
    rental.returnVehicle("M001");

    // Display available vehicles again
    rental.displayAvailableVehicles();

    return 0;
}
```