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

```
#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;
}
```

Revision #1
Created 2025-11-17 14:30:32 UTC by DS
Updated 2025-11-17 14:31:35 UTC by DS