

Object Oriented Programming C++

Chapter 6: Polymorphism

Coding Problems & Solutions — Set 3

Problems 18–25: Multilevel overriding, covariant returns, references, pure virtual destructors, ambiguity, and object slicing

Coding Problems & Solutions — Set 3

Problem 18: Multilevel Inheritance — Virtual Function Chaining

Write a program with three levels of inheritance (Vehicle → Car → SportsCar), where each level overrides the same virtual function describe(), and show that calling it through a Vehicle* always reaches the most-derived version.

```
#include <iostream>
using namespace std;
class Vehicle {
public:
    virtual void describe() { cout << "A generic vehicle" << endl; }
    virtual ~Vehicle() {}
};
class Car : public Vehicle {
public:
    void describe() override { cout << "A car with four wheels" << endl; }
};
class SportsCar : public Car {
public:
    void describe() override { cout << "A high-speed sports car" << endl; }
};
int main() {
    Vehicle *v = new SportsCar();
    v->describe();    // Reaches the most-derived override
    delete v;
}
```

Output:

A high-speed sports car

Explanation: Even though describe() passes through three levels of the hierarchy, dynamic binding always resolves to the version belonging to the object's actual (most-derived) type at run time — here SportsCar::describe() — regardless of how many intermediate classes overrode it along the way.

Problem 19: Covariant Return Types with Virtual Functions

Write a program where a base class Animal has a virtual function reproduce() returning Animal*, and a derived class Dog overrides it to return Dog* instead — demonstrating covariant return types.

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual Animal* reproduce() { cout << "New Animal born" << endl; return new Animal(); }
    virtual ~Animal() {}
};
class Dog : public Animal {
public:
    Dog* reproduce() override { cout << "New Dog born" << endl; return new Dog(); }
}
```

```
// covariant return
};
int main() {
    Animal *a = new Dog();
    Animal *baby = a->reproduce();
    delete baby; delete a;
}
```

Output:

New Dog born

Explanation: Normally an overriding function must have an IDENTICAL return type to the base version. C++ makes a special exception called a covariant return type: if the base returns `Animal*` (a pointer/reference to a class), the override is allowed to return a pointer/reference to a class derived from it, such as `Dog*`. This keeps overriding flexible while still preserving type safety.

Problem 20: Dynamic Binding Through References (Not Just Pointers)

Write a program showing that run-time polymorphism works equally well through a base class reference, not only through a base class pointer.

```
#include <iostream>
using namespace std;
class Instrument {
public:
    virtual void play() { cout << "Playing an instrument" << endl; }
    virtual ~Instrument() {}
};
class Piano : public Instrument {
public:
    void play() override { cout << "Playing the piano" << endl; }
};
void performOn(Instrument &ref) {    // takes a base class REFERENCE
    ref.play();
}
int main() {
    Piano p;
    performOn(p);    // reference bound to a Piano object
}
```

Output:

Playing the piano

Explanation: `performOn()` only knows about `Instrument&`, yet `ref.play()` still resolves dynamically to `Piano::play()` because the reference is bound to an actual Piano object. This proves dynamic binding depends on the object's real type, and works through references exactly as it does through pointers — as long as the parameter is a reference/pointer and not passed by value.

Problem 21: Pure Virtual Destructor

Write a program showing that a destructor CAN be declared pure virtual (to force a class to be abstract) while still requiring a definition, since destructors are always called during the destruction chain.

```
#include <iostream>
using namespace std;
class AbstractBase {
public:
    virtual ~AbstractBase() = 0;    // pure virtual destructor
};
AbstractBase::~~AbstractBase() {    // still MUST be defined
    cout << "AbstractBase destructor" << endl;
}
class Concrete : public AbstractBase {
public:
    ~Concrete() { cout << "Concrete destructor" << endl; }
};
int main() {
    AbstractBase *b = new Concrete();
    delete b;
}
```

Output:

```
Concrete destructor
AbstractBase destructor
```

Explanation: Declaring `~AbstractBase() = 0;` makes `AbstractBase` abstract (it cannot be instantiated) purely to prevent direct object creation, even though it may have no other pure virtual functions. But unlike an ordinary pure virtual function, a pure virtual destructor must still be defined outside the class, because every derived object's destruction chain will eventually call the base destructor — leaving it undefined would cause a linker error.

Problem 22: Overloading Ambiguity with Default Arguments

Write a program that demonstrates a compile-time ambiguity error caused by combining function overloading with default arguments, and then show the corrected version.

```
#include <iostream>
using namespace std;
class Printer {
public:
    void show(int a, int b = 10) { cout << "Two args: " << a << ", " << b << endl; }
    void show(int a) { cout << "One arg: " << a << endl; }    // AMBIGUOUS with show(int,int=10)
};
int main() {
    Printer p;
    p.show(5);    // Compile error: ambiguous call to 'show'
}
```

Output:

```
error: call of overloaded 'show(int)' is ambiguous
```

Explanation: Because `show(int, int=10)` can be called with just one argument (using the default), the compiler cannot decide between it and `show(int)` when `show(5)` is called — both are equally valid matches, producing an ambiguity error. The safe fix is to remove the default argument from the two-parameter overload, or give the overloads clearly distinct signatures, so each call has exactly one valid match.

```
// Fix: remove the default argument, forcing explicit calls
void show(int a, int b) { cout << "Two args: " << a << ", " << b << endl; }
void show(int a) { cout << "One arg: " << a << endl; }
// Now show(5) unambiguously calls show(int)
```

Problem 23: Polymorphic Payment System — Overloading and Overriding Combined

Write a program simulating a payment gateway where a base class `PaymentMethod` has a virtual `processPayment(double amount)`, overridden by `CardPayment` and `UpiPayment`, processed through a common function that takes a vector of base pointers.

```
#include <iostream>
#include <vector>
using namespace std;
class PaymentMethod {
public:
    virtual void processPayment(double amount) = 0;
    virtual ~PaymentMethod() {}
};
class CardPayment : public PaymentMethod {
public:
    void processPayment(double amount) override {
        cout << "Charged Rs." << amount << " to card (2% fee: " << amount * 0.02 <<
    )" << endl;
    }
};
class UpiPayment : public PaymentMethod {
public:
    void processPayment(double amount) override {
        cout << "Paid Rs." << amount << " via UPI (no fee)" << endl;
    }
};
void checkout(vector<PaymentMethod*> &methods, double amount) {
    for (auto *m : methods) m->processPayment(amount);
}
int main() {
    vector<PaymentMethod*> methods = { new CardPayment(), new UpiPayment() };
    checkout(methods, 1000);
    for (auto *m : methods) delete m;
}
```

Output:

```
Charged Rs.1000 to card (2% fee: 20)
Paid Rs.1000 via UPI (no fee)
```

Explanation: `checkout()` is written once against the abstract `PaymentMethod` interface, but works correctly for any current or future payment type. Each object's own `processPayment()` is invoked at run time through the vtable, so adding a new `WalletPayment` class later would need no change to `checkout()` itself — the central benefit of designing with pure virtual functions.

Problem 24: The Object Slicing Problem

Write a program that demonstrates object slicing — what happens when a derived class object is passed BY VALUE to a function expecting the base class, defeating polymorphism.

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() { cout << "Drawing a generic Shape" << endl; }
};
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing a Circle" << endl; }
};
void showShape(Shape s) {          // BY VALUE -- causes slicing
    s.draw();
}
void showShapeCorrect(Shape &s) {  // BY REFERENCE -- preserves polymorphism
    s.draw();
}
int main() {
    Circle c;
    showShape(c);                  // sliced: prints "generic Shape"
    showShapeCorrect(c);          // correct: prints "Circle"
}

```

Output:

```

Drawing a generic Shape
Drawing a Circle

```

Explanation: When Circle c is passed by value into showShape(Shape s), only the Shape portion of c is copied into a brand-new, plain Shape object — the Circle-specific part is “sliced off,” and the vptr now points to Shape's own vtable, so s.draw() calls Shape::draw(). Passing by reference (or pointer) instead avoids any copying, keeps the original object and its vptr intact, and lets polymorphism work as expected.