

Object Oriented Programming C++

Chapter 6: Polymorphism

Coding Problems & Solutions — Set 5

Problems 32–38: Advanced and allied topics — default-argument binding, operator variants, smart pointers, const overloading, static polymorphism, virtual inheritance, std::function

Coding Problems & Solutions — Set 5

This set moves slightly beyond the core syllabus points into closely related, exam-relevant extensions that commonly appear as “tricky output” or viva questions on this chapter.

Problem 32: Default Arguments in Virtual Functions Are Statically Bound

Write a program that demonstrates a subtle gotcha: the FUNCTION CALL in a virtual function is dynamically bound, but its DEFAULT ARGUMENT VALUES are resolved statically, using the pointer's declared type.

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void greet(string name = "Base-default") {
        cout << "Hello, " << name << endl;
    }
};
class Derived : public Base {
public:
    void greet(string name = "Derived-default") override {
        cout << "Hi, " << name << endl;
    }
};
int main() {
    Base *b = new Derived();
    b->greet();    // Which default value is used?
    delete b;
}
```

Output:

Hi, Base-default

Explanation: Which FUNCTION BODY runs (Derived::greet) is decided dynamically, at run time, based on the object's real type — as expected for a virtual function. But which DEFAULT ARGUMENT VALUE is substituted is decided statically, at compile time, based on the pointer's declared type (Base). This mismatch is a classic C++ trap: never rely on default arguments differing between a base and derived override, since the two decisions use different binding rules.

Problem 33: Overloading Prefix and Postfix Increment Operators

Write a program that overloads both the prefix (++obj) and postfix (obj++) increment operators for a Counter class, distinguished by a dummy int parameter.

```
#include <iostream>
using namespace std;
class Counter {
    int count;
public:
    Counter(int c = 0) : count(c) {}
    Counter& operator++() {                // prefix: ++obj
```

```

        ++count;
        return *this;
    }
    Counter operator++(int) {                // postfix: obj++ (dummy int marks it)
        Counter temp = *this;
        count++;
        return temp;
    }
    int value() { return count; }
};

int main() {
    Counter c(5);
    Counter afterPrefix = ++c;              // increments first, then copies
    Counter afterPostfix = c++;              // copies first, then increments
    cout << "afterPrefix: " << afterPrefix.value() << endl;
    cout << "afterPostfix: " << afterPostfix.value() << endl;
    cout << "final c: " << c.value() << endl;
}

```

Output:

```

afterPrefix: 6
afterPostfix: 6
final c: 7

```

Explanation: Both operators share the name `operator++`, so this is another form of function overloading — here the compiler tells them apart using a dummy, unused `int` parameter purely as a marker for the postfix version. Prefix returns the already-incremented object by reference; postfix must save a copy of the old value before incrementing, then return that saved copy.

Problem 34: Run-Time Polymorphism Using Smart Pointers

Write a program that achieves run-time polymorphism using `std::unique_ptr<Base>` instead of raw pointers, so that derived objects are destroyed automatically and safely.

```

#include <iostream>
#include <memory>
#include <vector>
using namespace std;
class Notification {
public:
    virtual void send() = 0;
    virtual ~Notification() {}
};
class SmsNotification : public Notification {
public:
    void send() override { cout << "Sending SMS" << endl; }
};
class PushNotification : public Notification {
public:
    void send() override { cout << "Sending Push Notification" << endl; }
};
int main() {
    vector<unique_ptr<Notification>> queue;
    queue.push_back(make_unique<SmsNotification>());
    queue.push_back(make_unique<PushNotification>());
    for (auto &n : queue) n->send();
    // No explicit delete needed -- unique_ptr destroys each object automatically,

```

```
// correctly calling the virtual destructor chain.
}
```

Output:

```
Sending SMS
Sending Push Notification
```

Explanation: `unique_ptr<Notification>` still relies on `Notification`'s virtual functions and virtual destructor to work correctly — polymorphism itself is unchanged. What smart pointers add is automatic, exception-safe cleanup: when the vector goes out of scope, each `unique_ptr` automatically calls `delete` on its managed object, invoking the proper (virtual) destructor chain without the programmer writing a single `delete` statement.

Problem 35: Overloading Based on const-ness of the Object

Write a program that overloads a member function twice — once as `const` and once as non-`const` — so that the compiler picks a different version depending on whether the calling object is `const`.

```
#include <iostream>
using namespace std;
class DataBox {
    int value;
public:
    DataBox(int v) : value(v) {}
    void show() { cout << "Non-const show(): " << value << " (modifiable)" << endl; }
    void show() const { cout << "Const show(): " << value << " (read-only)" << endl; }
};
int main() {
    DataBox box(10);
    const DataBox constBox(20);
    box.show();           // calls the non-const version
    constBox.show();      // calls the const version
}
```

Output:

```
Non-const show(): 10 (modifiable)
Const show(): 20 (read-only)
```

Explanation: The `const`-qualification of a member function is part of its signature for overload-resolution purposes, just like a parameter type. When the calling object itself is `const` (`constBox`), only `const` member functions are callable on it, so the compiler is forced to choose `show() const`; for a non-`const` object, the non-`const` overload is preferred when both are viable.

Problem 36: Static Polymorphism Using Templates (CRTP) as an Alternative to Virtual Functions

Write a program that achieves polymorphism-like behaviour at COMPILE TIME using the Curiously Recurring Template Pattern (CRTP), avoiding the run-time cost of virtual function calls.

```
#include <iostream>
using namespace std;
template <typename Derived>
class Shape {
public:
```

```

    void draw() {
        static_cast<Derived*>(this)->drawImpl();    // compile-time dispatch
    }
};
class Circle : public Shape<Circle> {
public:
    void drawImpl() { cout << "Drawing Circle (CRTP)" << endl; }
};
class Square : public Shape<Square> {
public:
    void drawImpl() { cout << "Drawing Square (CRTP)" << endl; }
};
template <typename T>
void render(Shape<T> &s) { s.draw(); }
int main() {
    Circle c; Square sq;
    render(c);
    render(sq);
}

```

Output:

```

Drawing Circle (CRTP)
Drawing Square (CRTP)

```

Explanation: CRTP achieves a similar effect to virtual functions — calling the “right” drawImpl() for each shape — but the exact type is known and resolved entirely at COMPILE TIME through templates, with no vtable, no vptr, and no run-time lookup cost. The trade-off is flexibility: unlike true virtual dispatch, a single Shape<T>* cannot point to different shape types at run time, since each instantiation is its own distinct type.

Problem 37: Virtual Base Classes — Resolving the Diamond Ambiguity

Write a program showing the classic diamond inheritance problem, and how declaring the shared base as virtual removes the resulting ambiguity when a common function or member is accessed.

```

#include <iostream>
using namespace std;
class Device {
public:
    void powerOn() { cout << "Device powered on" << endl; }
};
class Printer : virtual public Device {};    // virtual inheritance
class Scanner : virtual public Device {};    // virtual inheritance
class AllInOnePrinter : public Printer, public Scanner {};
int main() {
    AllInOnePrinter aio;
    aio.powerOn();    // Unambiguous: only ONE Device sub-object exists
}

```

Output:

```

Device powered on

```

Explanation: Without the virtual keyword on Printer and Scanner's inheritance from Device, AllInOnePrinter would contain TWO separate Device sub-objects (one via Printer, one via Scanner), making aio.powerOn() ambiguous and a compile error. Declaring the inheritance as virtual public Device ensures both paths share a

single common Device sub-object, resolving the diamond ambiguity so powerOn() can be called directly and unambiguously.

Problem 38: std::function as a Lightweight Alternative to Virtual Functions

Write a program that selects different behaviours at run time using std::function instead of a class hierarchy with virtual functions, useful for simple cases needing polymorphic-like behaviour without inheritance.

```
#include <iostream>
#include <functional>
#include <vector>
using namespace std;
void applyDiscount(double amount, function<double(double)> strategy) {
    cout << "Final amount: " << strategy(amount) << endl;
}
int main() {
    auto noDiscount = [](double a) { return a; };
    auto tenPercent = [](double a) { return a * 0.90; };
    auto twentyPercent = [](double a) { return a * 0.80; };
    vector<function<double(double)>> strategies = { noDiscount, tenPercent,
twentyPercent };
    for (auto &s : strategies)
        applyDiscount(1000, s);
}
```

Output:

```
Final amount: 1000
Final amount: 900
Final amount: 800
```

Explanation: Here, the “which behaviour runs” decision — the essence of polymorphism — is made by storing and calling different callable objects (lambdas) through std::function, rather than through a base class and overridden virtual functions. This is a common modern C++ (and interview-relevant) alternative for simple cases: it avoids writing a whole class hierarchy just to select between a handful of interchangeable behaviours.