

Object Oriented Programming C++

Chapter 6: Polymorphism

Coding Problems & Solutions — Set 4

Problems 25–31: NVI idiom, final specifier, dynamic_cast, multiple inheritance, name hiding, array slicing, stream operators

Coding Problems & Solutions — Set 4

Problem 25: Non-Virtual Interface (NVI) Idiom

Write a program where a public non-virtual function in the base class controls the algorithm's structure, while calling a private virtual function that derived classes override to customise just one step.

```
#include <iostream>
using namespace std;
class Report {
public:
    void generate() {                // public, NOT virtual -- fixed template
        cout << "--- Report Start ---" << endl;
        writeBody();                // customizable step
        cout << "--- Report End ---" << endl;
    }
    virtual ~Report() {}
private:
    virtual void writeBody() { cout << "Generic body" << endl; }
};
class SalesReport : public Report {
private:
    void writeBody() override { cout << "Sales figures for Q3" << endl; }
};
int main() {
    Report *r = new SalesReport();
    r->generate();                // structure fixed, body customized
    delete r;
}
```

Output:

```
--- Report Start ---
Sales figures for Q3
--- Report End ---
```

Explanation: The Non-Virtual Interface idiom separates the public, stable interface (`generate()`, non-virtual so it cannot be accidentally overridden) from the customizable implementation detail (`writeBody()`, private and virtual). Even though `writeBody()` is private in the base class, a derived class can still override it — access control and virtuality are independent in C++ — and `generate()` still calls the correct overridden version dynamically.

Problem 26: The 'final' Specifier — Preventing Further Overriding

Write a program showing how the final specifier stops a virtual function from being overridden any further down the inheritance chain, and what the compiler does if that rule is broken.

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void process() { cout << "Base process" << endl; }
};
class Middle : public Base {
```

```

public:
    void process() override final { cout << "Middle process (locked)" << endl; }
};
class Bottom : public Middle {
public:
    // void process() override { cout << "Bottom process"; } // ERROR if
    // uncommented:
    // "declaration of 'process' overrides a 'final' function"
};
int main() {
    Base *b = new Bottom();
    b->process(); // still calls Middle::process(), since Bottom does not
    // override it
    delete b;
}

```

Output:

Middle process (locked)

Explanation: Marking `Middle::process()` as `final` tells the compiler that no class derived from `Middle` may override `process()` again; attempting to do so in `Bottom` is a compile-time error. This is useful when a class wants to guarantee that a particular piece of behaviour can never be changed further down an inheritance chain, even though the function is otherwise virtual.

Problem 27: Safe Downcasting with `dynamic_cast`

Write a program that safely converts a base class pointer back to a specific derived class pointer using `dynamic_cast`, and correctly detects when the conversion is not valid.

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() { cout << "Drawing a shape" << endl; }
    virtual ~Shape() {}
};
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing a circle" << endl; }
    void specialCircleMethod() { cout << "Circle-only feature used" << endl; }
};
class Square : public Shape {
public:
    void draw() override { cout << "Drawing a square" << endl; }
};
int main() {
    Shape *s = new Circle();
    Circle *c = dynamic_cast<Circle*>(s);
    if (c) c->specialCircleMethod(); // succeeds: s really points to a Circle
    Shape *sq = new Square();
    Circle *wrong = dynamic_cast<Circle*>(sq);
    if (!wrong) cout << "Cast failed: sq is not a Circle" << endl;
    delete s; delete sq;
}

```

Output:

Circle-only feature used
Cast failed: sq is not a Circle

Explanation: `dynamic_cast` checks the actual run-time type of the object (using the same polymorphic type information that supports virtual functions) and returns a null pointer if the conversion is not valid, instead of causing undefined behaviour. This lets a program safely use a derived-class-only method after first confirming the object's true type, and `dynamic_cast` only works on classes with at least one virtual function (polymorphic classes).

Problem 28: Overriding with Multiple Base Classes

Write a program where a class inherits from two base classes that each declare a virtual function with the same name, and show how the derived class overrides both with a single function.

```
#include <iostream>
using namespace std;
class Flyer {
public:
    virtual void move() { cout << "Flying" << endl; }
};
class Swimmer {
public:
    virtual void move() { cout << "Swimming" << endl; }
};
class Duck : public Flyer, public Swimmer {
public:
    void move() override {          // overrides BOTH base versions at once
        cout << "Duck waddles, but can also: ";
        Flyer::move();
    }
};
int main() {
    Duck d;
    Flyer *f = &d;
    Swimmer *s = &d;
    f->move();    // Duck::move() via Flyer pointer
    s->move();    // Duck::move() via Swimmer pointer
}
```

Output:

Duck waddles, but can also: Flying
Duck waddles, but can also: Flying

Explanation: `Duck::move()` simultaneously overrides `Flyer::move()` and `Swimmer::move()`, since it matches both signatures. Calling `move()` through either a `Flyer*` or a `Swimmer*` that actually points to a `Duck` correctly dispatches to the single `Duck::move()` — and inside it, `Flyer::move()` is invoked explicitly using scope resolution to reuse specific base behaviour.

Problem 29: Overriding Mistake — Forgetting the virtual Keyword (Name Hiding)

Write a program that shows what happens when a base class function is NOT declared virtual, and a derived class defines a function with the same signature — this is name hiding, not true overriding.

```
#include <iostream>
using namespace std;
class Base {
public:
    void greet() { cout << "Hello from Base" << endl; }    // NOT virtual
};
class Derived : public Base {
public:
    void greet() { cout << "Hello from Derived" << endl; }    // hides Base::greet,
    does NOT override it
};
int main() {
    Base *b = new Derived();
    b->greet();        // calls Base::greet() -- static binding, NOT polymorphic!
    Derived d;
    d.greet();        // calls Derived::greet() directly on the object
    delete b;
}
```

Output:

```
Hello from Base
Hello from Derived
```

Explanation: Since greet() is not virtual in Base, the compiler resolves b->greet() using the POINTER's declared type (Base) at compile time, ignoring the fact that b actually points to a Derived object. This is a very common exam trap: without the virtual keyword, a derived class function of the same name only hides the base version rather than overriding it, and true run-time polymorphism does not occur.

Problem 30: Polymorphism Failure with an Array of Objects (Not Pointers)

Write a program that shows polymorphism does NOT work when derived class objects are stored directly in an array of the base type, due to object slicing.

```
#include <iostream>
using namespace std;
class Employee {
public:
    virtual void role() { cout << "Generic Employee" << endl; }
};
class Manager : public Employee {
public:
    void role() override { cout << "Manager" << endl; }
};
int main() {
    Employee staff[2];        // array of BASE OBJECTS, not pointers
    staff[0] = Manager();    // Manager gets sliced down to Employee when assigned
    staff[0].role();        // prints "Generic Employee", NOT "Manager"
    Employee *staffPtrs[1] = { new Manager() };
    staffPtrs[0]->role();    // prints "Manager" -- correct, using pointers
    delete staffPtrs[0];
}
```

Output:

```
Generic Employee
Manager
```

Explanation: An Employee staff[2] array can only ever hold true Employee objects; assigning a Manager() to staff[0] copies just the Employee portion, slicing away everything specific to Manager (including its vptr, which is reset to point at Employee's own vtable). Polymorphism through arrays or containers therefore requires an array/container of BASE CLASS POINTERS (or references), never an array of base class objects by value.

Problem 31: Overloading Stream Operators with a Virtual Display Function

Write a program that overloads operator<< using a friend function which internally calls a virtual display() function, so that cout << basePtr; prints the correct derived class's data.

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void display(ostream &out) const { out << "Shape (generic)"; }
    friend ostream& operator<<(ostream &out, const Shape &s) {
        s.display(out);          // dispatches dynamically even though operator<<
        itself isn't virtual
        return out;
    }
    virtual ~Shape() {}
};
class Square : public Shape {
    double side;
public:
    Square(double s) : side(s) {}
    void display(ostream &out) const override {
        out << "Square with side " << side;
    }
};
int main() {
    Shape *s = new Square(5);
    cout << *s << endl;    // uses operator<<, which calls the overridden display()
    delete s;
}
```

Output:

Square with side 5

Explanation: Operators like << cannot themselves be declared virtual, but this is worked around by having the (non-virtual) friend operator<< call a virtual helper function, display(), on the object. Since display() is virtual, it still dispatches to Square::display() at run time even though the operator overload code is written only once, in the base class.