

Object Oriented Programming C++

Chapter 6: Polymorphism

Coding Problems & Solutions

Covering: Overloading, Overriding, Virtual Functions, Static/Dynamic Binding, Pure Virtual Functions, Abstract Classes, Virtual Destructors

Coding Problems & Solutions

Problem 1: Function Overloading — Area Calculator

Write a program to overload a function `area()` to compute the area of a square (1 argument), a rectangle (2 arguments), and a circle (1 double argument using a different type).

```
#include <iostream>
using namespace std;
class AreaCalc {
public:
    int area(int side) { return side * side; }           // square
    int area(int l, int b) { return l * b; }           // rectangle
    double area(double r) { return 3.14159 * r * r; }   // circle
};
int main() {
    AreaCalc a;
    cout << "Square: "    << a.area(5) << endl;
    cout << "Rectangle: " << a.area(4, 6) << endl;
    cout << "Circle: "    << a.area(3.0) << endl;
}
```

Output:

```
Square: 25
Rectangle: 24
Circle: 28.2743
```

Explanation: The compiler picks the correct `area()` overload purely from the number and type of arguments at compile time — int for square, two ints for rectangle, and double for circle. This is compile-time (static) polymorphism.

Problem 2: Operator Overloading — Vector Addition

Write a program to overload the '+' operator to add two 2D vectors (points), and the '<<' operator to display them.

```
#include <iostream>
using namespace std;
class Vector2D {
    float x, y;
public:
    Vector2D(float x = 0, float y = 0) : x(x), y(y) {}
    Vector2D operator+(const Vector2D &v) {
        return Vector2D(x + v.x, y + v.y);
    }
    friend ostream& operator<<(ostream &out, const Vector2D &v) {
        out << "(" << v.x << ", " << v.y << ")";
        return out;
    }
};
int main() {
    Vector2D v1(2, 3), v2(4, 1);
    Vector2D v3 = v1 + v2;
```

```
    cout << "Result: " << v3;
}
```

Output:

Result: (6, 4)

Explanation: operator+ is overloaded as a member function taking a Vector2D, while operator<< is overloaded as a friend function so it can take an ostream on the left side. Both are resolved at compile time based on operand types.

Problem 3: Function Overriding — Bank Account Interest

Write a program where a base class Account has a virtual function calculateInterest(), overridden differently in SavingsAccount and FixedDeposit.

```
#include <iostream>
using namespace std;
class Account {
protected:
    float balance;
public:
    Account(float b) : balance(b) {}
    virtual float calculateInterest() { return balance * 0.02; }
};
class SavingsAccount : public Account {
public:
    SavingsAccount(float b) : Account(b) {}
    float calculateInterest() override { return balance * 0.04; }
};
class FixedDeposit : public Account {
public:
    FixedDeposit(float b) : Account(b) {}
    float calculateInterest() override { return balance * 0.07; }
};
int main() {
    Account *acc;
    SavingsAccount sa(10000);
    FixedDeposit fd(10000);
    acc = &sa; cout << "Savings interest: " << acc->calculateInterest() << endl;
    acc = &fd; cout << "FD interest: " << acc->calculateInterest() << endl;
}
```

Output:

Savings interest: 400
FD interest: 700

Explanation: Both derived classes override the same virtual function signature. Since the base pointer acc calls calculateInterest() dynamically, the correct overridden version runs based on the object actually pointed to — this is run-time polymorphism.

Problem 4: Overloaded Virtual Functions in a Base Class

Write a program where a base class declares two overloaded virtual print() functions, and a derived class overrides only one of them while still being able to call both.

```

#include <iostream>
using namespace std;
class Printer {
public:
    virtual void print(int x) { cout << "Base int: " << x << endl; }
    virtual void print(string s) { cout << "Base string: " << s << endl; }
};
class MyPrinter : public Printer {
public:
    using Printer::print;    // brings both overloads into scope
    void print(int x) override { cout << "Derived int: " << x << endl; }
};
int main() {
    Printer *p = new MyPrinter();
    p->print(10);            // Derived int: 10
    p->print(string("Hi"));  // Base string: Hi (inherited, not hidden)
}

```

Output:

```

Derived int: 10
Base string: Hi

```

Explanation: Without the using-declaration, `MyPrinter::print(int)` would hide ALL base class `print()` overloads (C++ name-hiding rule), making `print(string)` inaccessible on a `MyPrinter` object/pointer. The `using Printer::print;` line brings the hidden overload back into scope.

Problem 5: Static vs Dynamic Binding Demonstration

Write a program that calls one non-virtual and one virtual function through a base class pointer, to clearly show the difference between static and dynamic binding.

```

#include <iostream>
using namespace std;
class Base {
public:
    void staticFn() { cout << "Base::staticFn (static binding)" << endl; }
    virtual void dynamicFn() { cout << "Base::dynamicFn (dynamic binding)" << endl; }
};
class Derived : public Base {
public:
    void staticFn() { cout << "Derived::staticFn (static binding)" << endl; }
    void dynamicFn() override { cout << "Derived::dynamicFn (dynamic binding)" << endl; }
};
int main() {
    Base *bptr = new Derived();
    bptr->staticFn();    // resolved at compile time using pointer type
    bptr->dynamicFn();  // resolved at run time using object type
}

```

Output:

```

Base::staticFn (static binding)
Derived::dynamicFn (dynamic binding)

```

Explanation: staticFn() is not virtual, so the compiler binds the call based on the pointer's declared type (Base) — static binding. dynamicFn() is virtual, so the call is resolved at run time using the vtable of the actual object (Derived) — dynamic binding.

Problem 6: Pure Virtual Function — Abstract Employee Hierarchy

Write a program using an abstract class Employee with a pure virtual function calculateSalary(), implemented differently by Manager and Developer.

```
#include <iostream>
using namespace std;
class Employee {
public:
    virtual float calculateSalary() = 0;    // pure virtual
    virtual ~Employee() {}
};
class Manager : public Employee {
public:
    float calculateSalary() override { return 60000; }
};
class Developer : public Employee {
public:
    float calculateSalary() override { return 45000; }
};
int main() {
    Employee *e[2];
    e[0] = new Manager();
    e[1] = new Developer();
    for (int i = 0; i < 2; i++)
        cout << "Salary: " << e[i]->calculateSalary() << endl;
    delete e[0]; delete e[1];
}
```

Output:

```
Salary: 60000
Salary: 45000
```

Explanation: Employee cannot be instantiated (Employee e; would be a compile error) because it has a pure virtual function. An array of Employee* pointers is used instead, and each element correctly dispatches to its own overridden calculateSalary() at run time.

Problem 7: Virtual Constructor Idiom — Document Cloning System

Write a program simulating a document management system where clone() acts as a virtual constructor to duplicate documents of different derived types without knowing their exact type.

```
#include <iostream>
using namespace std;
class Document {
public:
    virtual Document* clone() = 0;
    virtual void show() = 0;
    virtual ~Document() {}
};
```

```

class Invoice : public Document {
public:
    Document* clone() override { return new Invoice(*this); }
    void show() override { cout << "Invoice Document" << endl; }
};
class Report : public Document {
public:
    Document* clone() override { return new Report(*this); }
    void show() override { cout << "Report Document" << endl; }
};
void duplicateAndShow(Document *d) {
    Document *copy = d->clone();    // acts like a "virtual constructor"
    copy->show();
    delete copy;
}
int main() {
    Document *d1 = new Invoice();
    Document *d2 = new Report();
    duplicateAndShow(d1);
    duplicateAndShow(d2);
    delete d1; delete d2;
}

```

Output:

```

Invoice Document
Report Document

```

Explanation: duplicateAndShow() receives only a Document* and has no idea whether it points to an Invoice or a Report, yet clone() still creates a new object of the correct derived type. This is the Virtual Constructor Idiom — C++ has no literal virtual constructor, but this pattern achieves the same effect.

Problem 8: Virtual Destructor — Preventing Memory Leaks

Write a program that shows what goes wrong when a base class destructor is NOT virtual, and then fix it.

```

#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base constructed" << endl; }
    ~Base() { cout << "Base destroyed" << endl; }    // NOT virtual (buggy version)
};
class Derived : public Base {
    int *buffer;
public:
    Derived() { buffer = new int[100]; cout << "Derived constructed" << endl; }
    ~Derived() { delete[] buffer; cout << "Derived destroyed" << endl; }
};
int main() {
    Base *b = new Derived();
    delete b;    // Undefined behaviour: Derived::~~Derived() is skipped, buffer
leaks
}

```

Output:

```

Base constructed
Derived constructed

```

Base destroyed
(Derived destroyed is NEVER printed – buffer leaks)

Explanation: Because `~Base()` is not virtual, `delete b` only calls the destructor matching the static (pointer) type, Base. The fix is to declare virtual `~Base() { ... }` in the base class — then `delete b` correctly calls `~Derived()` first (freeing buffer) and then `~Base()`, in the proper order.

```
// Fix: declare the base destructor virtual
class Base {
public:
    virtual ~Base() { cout << "Base destroyed" << endl; }
};
// Now: delete b -> "Derived destroyed" then "Base destroyed", no leak
```

Problem 9: Shape Hierarchy — Combining Overloading, Overriding and Abstract Class

Write a comprehensive program combining an abstract Shape class (pure virtual `area()` and `perimeter()`), an overloaded constructor, and run-time polymorphism through a vector of shape pointers.

```
#include <iostream>
#include <vector>
using namespace std;
class Shape {
public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
    virtual ~Shape() {}
};
class Rectangle : public Shape {
    double l, w;
public:
    Rectangle(double side) : l(side), w(side) {}           // overloaded ctor: square
    Rectangle(double l, double w) : l(l), w(w) {}         // overloaded ctor:
rectangle
    double area() override { return l * w; }
    double perimeter() override { return 2 * (l + w); }
};
int main() {
    vector<Shape*> shapes;
    shapes.push_back(new Rectangle(4));                    // square 4x4
    shapes.push_back(new Rectangle(3, 6));                 // rectangle 3x6
    for (Shape *s : shapes)
        cout << "Area: " << s->area() << ", Perimeter: " << s->perimeter() << endl;
    for (Shape *s : shapes) delete s;
}
```

Output:

Area: 16, Perimeter: 16
Area: 18, Perimeter: 18

Explanation: This problem layers all the chapter's ideas together: the Rectangle constructor is overloaded (compile-time polymorphism), `area()/perimeter()` are pure virtual in the abstract Shape base (forcing every derived class to implement them), and the loop calls them through `Shape*` pointers, with the correct overridden version selected at run time for each object.